J.-J.
LABARTHE

BASIC 1000D

ATARI ST

LOGICIEL

MANUEL DE
REFERENCE

LOGICIEL

BASIC 1000D

ATARI ST

MANUEL DE REFERENCE

Jean-Jacques LABARTHE

Laboratoire Aimé Cotton

Université de Paris XI

1990

Table des matières

1

Présentation du Basic 1000d



Performances mathématiques

Basic 1000d est un langage de programmation dont la précision des calculs numériques approchés en nombres réels ou complexes dépasse 1000 chiffres. Il permet aussi de calculer de façon exacte, sans approximation, avec des entiers ayant jusqu'à 19723 chiffres, et de manipuler les expressions algébriques littérales.

L'apprentissage de ce langage sera rapide pour ceux qui connaissent déjà un Basic. En effet sa programmation est très proche des Basics usuels.

Les performances extraordinaires en analyse numérique et en algèbre font du Basic 1000d un outil sans égal sur micro-ordinateurs, destiné aux chercheurs des laboratoires, centres de recherches et bureaux d'études, aux amateurs d'algèbre et d'arithmétique, aux lycéens, étudiants et professeurs.

Calculs en flottants

Le programme suivant calcule $\sqrt{2}$ avec 50 chiffres exacts :

```
precision 50
print "sqr(2)=";sqr(2)
```

Sortie (115 ms)

```
sqr(2)=  0.14142135623730950488016887242096980785696718753769~ E+1
```

L'indication 115 ms après Sortie indique le temps du calcul de l'exemple, en millisecondes. Le même calcul est tout aussi facile avec 1000 chiffres, mais avec cette précision le programme dure 4 secondes.

Voici un exemple de calcul en nombres complexes, avec 25 chiffres exacts.

```
precision 25
complex i
print exp(i)
```

Sortie (500 ms)

```
0.5403023058681397174009366~ +i*0.8414709848078965066525023~
```

Calculs exacts en entiers et rationnels

Comptons exactement les grains de l'échiquier $(1 + 2 + 4 + \cdots + 2^{63})$

```
print sum(i=1,64 of 2^(i-1))
```

Sortie (285 ms)

```
18446744073709551615
```

Le programme suivant utilise le processeur rationnel du Basic 1000d.

```
print (2^30000+1/3)-2^30000
```

Sortie (135 ms)

```
1/3
```

Les nombres rationnels m/n avec m et n entiers sont représentés exactement et les opérations sont effectuées sans aucune approximation. Dans

l'exemple, le nombre $2^{30000} + 1/3$ est d'abord calculé (il occupe 2762 octets en mémoire), puis le nombre 2^{30000} est calculé et retranché au nombre précédent.

Les procédés modernes de communication, tant pour la vérification des données que pour la transmission d'informations confidentielles, font en général appel à des méthodes arithmétiques de codage. De telles méthodes utilisent des grands nombres, des nombres premiers, et des calculs modulaires. En Basic 1000d ces calculs sont aussi simples que le calcul de $2 + 2$. Voici trois exemples dans ce domaine.

Calcul d'un nombre premier aléatoire de 25 chiffres :

```
print prime(10^24+random(9*10^24))
```

Sortie (12525 ms)

```
9691693562453961335247239
```

Calcul modulo 37 du nombre $8888^{10^{777}}$:

```
print mdpwre(8888,10^777,37)
```

Sortie (8520 ms)

```
26
```

Factorisation d'un nombre aléatoire :

```
w=random(10^10)
print "Factorisation de w=";w
print prfact$(w)
```

Sortie (175 ms)

```
Factorisation de w=  4202284092
 2^2 * 3 * 23 * 29 * 163 * 3221
```

Calcul formel

Le programme suivant est un exemple de calcul formel.

```
W=(1+X)^2
PRINT W
```

Sortie (35 ms)

```
X^2 +2*X +1
```

En Basic ordinaire, `X` et `W` sont des variables qui prennent certaines valeurs. Par exemple si `X` vaut 0, alors `W` vaut 1. Le même programme écrit en Basic 1000d est très différent. Le nom `X` qui apparaît à droite dans la première ligne est un littéral. Il n'a pas de valeur. `W` qui apparaît à gauche est une variable. Sa valeur est l'expression `(1+X)^2`. Le résultat du `PRINT` est la forme développée de l'expression.

Basic 1000d est capable de traiter les expressions rationnelles écrites avec plusieurs littéraux, par exemple pour les développer comme dans :

```
print formd( (1+x/5)^2/(a-b)^2 )
```

Sortie (110 ms)

```
1/25* [x^2 +10*x +25]* [a^2 -2*a*b +b^2]^-1
```

Dans le problème inverse (factorisation), Basic 1000d est très brillant (il peut trouver tous les facteurs irréductibles sur **Q**).

```
print "1+x^9 =";formf(1+x^9)
```

Sortie (1090 ms)

```
1+x^9 =   [x +1]* [x^6 -x^3 +1]* [x^2 -x +1]
```

Basic 1000d peut effectuer substitutions, dérivations. Par exemple la dérivée de $1/(ax+b)^{12}$ est calculée par le programme

```
print der(1/(a*x+b)^12,x)
```

Sortie (2335 ms)

```
-12* [a]* [a*x +b]^-13
```

Voici quelques autres exemples des possibilités du Basic 1000d.

Calcul de l'intégrale :

$$\int \frac{2x^2+6x+5}{x^4+6x^3+13x^2+12x+4}dx$$

```
print intg([2*x^2+6*x+5]/[x^4+6*x^3+13*x^2+12*x+4])
```

Sortie (505 ms)

```
- [2*x +3]* [x +2]^-1* [x +1]^-1
```

Développement limité au 7ème ordre au voisinage de zéro de $\sin(\tan x)$

```
print "sin(tg(x))=";str$( ssin(taylor(ssin(x,7)/scos(x,
 7),7),7),/x);"  + ..."
```

Sortie (785 ms)

```
sin(tg(x))= (  1)*x+(  1/6)*x^3+( -1/40)*x^5+( -55/1008)*x^7  + ...
```

Calcul du déterminant d'ordre 5 :

$$\begin{vmatrix} a & b & 1/4 & 1/5 & 1/6 \\ c & d & 2/5 & 1/3 & 2/7 \\ 3/4 & 3/5 & 1/2 & 3/7 & 3/8 \\ 4/5 & 2/3 & 4/7 & 1/2 & 4/9 \\ 5/6 & 5/7 & 5/8 & 5/9 & 1/2 \end{vmatrix}$$

```
var MAT(5,5)
for i=1,5
  for j=1,5
    MAT(i,j)=i/(i+j)
  next j
next i
MAT(1,1)=a
MAT(1,2)=b
MAT(2,1)=c
MAT(2,2)=d
print det(MAT,5)
```

Sortie (1985 ms)

```
 1/508032*a*d -17/17287200*a -1/508032*b*c +29/22226400*b +29/4445280
0*c -1/1058400*d +11/277830000
```

Résolution du système d'équations :

$$\begin{cases} x+y=10 \\ xy=16 \end{cases}$$

Sa résolution s′obtient en éliminant y entre les deux équations puis en factorisant le polynôme obtenu. La seule instruction suivante suffit.

```
print formf(elim(x+y-10,x*y-16,y))
```

Sortie (195 ms)

```
- [x -8]* [x -2]
```

De ce résultat on tire de suite les solutions $x = 8, y = 2$ et $x = 2, y = 8$.

Calculs en expressions complexes, en imposant des conditions. L'exemple suivant calcule $\cos 3x$ en fonction de $\cos x$. Pour cela les deux littéraux, `c` et `s` représentant $\cos x$ et $\sin x$, sont liés par la relation ($\sin^2 x + \cos^2 x = 1$) `s^2+c^2=1`. Une expression pour $\cos 3x$ est alors obtenue comme partie réelle de e^{3ix} qui se calcule par `(c+i*s)^3`.

```
cond s^2+c^2-1,s
complex i
wexpr=(c+i*s)^3
print "cos(3x)=";change$(Re(wexpr),"c","cos(x)")
```

Sortie (175 ms)

```
cos(3x)=  4*cos(x)^3 -3*cos(x)
```

Les performances mentionnées jusqu'ici sont celles des fonctions de base du Basic 1000d. Par programmation, les possibilités du Basic peuvent encore être étendues. Toutes les fonctions nécessaires au décorticage d'une expression (trouver les littéraux, les exposants, les coefficients, etc.) sont présentes et permettent de programmer soi-même n'importe quelle manipulation d'expressions non prévue dans Basic 1000d.

On trouvera dans la bibliothèque MATH de nombreuses extensions, comme par exemple le traitement algébrique exact des expressions trigonométriques. Une autre extension, décrite dans ce mode d'emploi, permet les calculs exacts sur des nombres entiers de plus de 200000 chiffres.

Programmation

Les procédures et fonctions sont d'une souplesse d'emploi remarquable. L'exemple suivant calcule et écrit les 1000 premiers nombres de la suite de Fibonacci. Basic 1000d permet les appels récursifs (la fonction `fibo` s'appelle elle-même). Avec la commande `remember`, qui mémorise les dernières valeurs calculées, le temps de calcul de `fibo`(n) est proportionnel à n. Sans cette commande, le temps de calcul devenant exponentiel en n, il serait impossible d'aller jusqu'à `fibo`(1000) par récurrence comme ici.

```
for i=1 to 1000
  print "FIBO(";i;")=";fibo(i)
```

```
          next i
          stop
     fibo:function(x)
          remember x
          if x<2
            value=x
          else
            value=fibo(x-1)+fibo(x-2)
          endif
          return
```

Sortie (232 s)

```
 FIBO(  1)=  1
 FIBO(  2)=  1
 FIBO(  3)=  2
 FIBO(  4)=  3
 FIBO(  5)=  5
 ...
 FIBO(  1000)=  434665576869374564356885276750406258025646605173717804O
  2481729089536555417949051890403879840079255169295922593080322634775208
  9689623239873322471161642996440906533187938298969649928516003704476137
  795166849228875
```

Voici un exemple de procédure.

```
          Etoile 125,200
          stop
   Etoile:procedure(x,y)
          local index i
          origin x,y
          plot 50,0
          for i=0 to 7
            line to 50*cos(6*pi*i/7),50*sin(6*pi*i/7)
          next i
          return
```

Le programme ci-dessus, dessine une étoile. L'appel de la procédure `Etoile` se fait de façon analogue à l'appel des commandes internes du Basic, par exemple

```
          Etoile 125,60
```

trace l'étoile au point 125,60.

En Basic 1000d la transmission des arguments à un sous-programme peut se faire par valeur, par adresse ou par nom. Dans l'exemple suivant on transmet à la procédure `op` le signe d'une opération, qui est utilisé une première fois pour l'écrire et une deuxième fois pour effectuer une opération.

```
          op +
          op -
          op /
```

```
        op *
        op ^
        stop
     op:procedure
        print "164@1 10=";164 @1 10
        return
```

Sortie (150 ms)

```
164+10=  174
164-10=  154
164/10=  82/5
164*10=  1640
164^10=  14074678408802364030976
```

Le nombre d'arguments peut être variable dans un sous-programme donné. Dans l'exemple suivant la fonction `pw` calcule le carré de son argument, s'il y en a un seul, ou bien l'élève à la puissance donnée par le deuxième argument.

```
        print pw(7);pw(7,3)
        stop
     pw:function(x)
        if @0=1
          value=x^2
        else
          value=x^(@2)
        endif
        return
```

Sortie (50 ms)

```
  49  343
```

Basic 1000d a été muni d'instructions spéciales en vue de faciliter la création et le traitement de nouveaux objets mathématiques. Le traitement d'expressions trigonométriques de la bibliothèque Math, déjà mentionné, utilise ces possibilités. Dans l'exemple suivant la commande `xqt` permet d'exécuter un programme en Basic donné sous forme de chaîne de caractères.

```
        c$="for i=1 to "& random(10) æ
        xqt c$ &"print i;"æ &"next"
```

Sortie (55 ms)

```
  1  2  3  4  5
```

On trouvera dans le Basic 1000d toutes les fonctions et commandes des Basics modernes, y compris les fonctions graphiques et musicales, et tous les appels du système. La programmation par structures, comme dans le langage Pascal, est également bien représentée en Basic 1000d.

Basic 1000d est capable d'absorber un ensemble de programmes, qui rendus invisibles, se comportent comme de nouvelles fonctions et commandes du Basic. Chacun peut ainsi se constituer des bibliothèques adaptées à ses propres besoins. Les nombreuses instructions de décodage du Basic 1000d rendent même possible l'écriture de véritables émulateurs d'autres langages.

Le Basic 1000d permet l'écriture et l'accès (par Help) à des modes d'emploi ou aide-mémoire. Bientôt vous explorerez le menu DEBUG. C'est un débogueur très sophistiqué (pas à pas, points d'arrêt, sortie de boucle, retour de procédure, valeur d'une variable obtenue par simple click sur le nom de la variable, etc.) qui rend la mise au point des programmes incomparablement plus aisée et rapide que le mode TRACE des Basics usuels.

Installer une application

La procédure suivante permet de lancer un programme en Basic à partir du bureau Gem, par simple double cliquage sur le nom du fichier source. Elle permet également le chargement du Basic et d'un fichier d'aide ou bibliothèque par double cliquage sur ce fichier. Copier sur une disquette, le fichier 1000d.prg et les sources en Basic des programmes d'application (d'extension Z). Booter le bureau, ouvrir la fenêtre de cette disquette et effectuer un simple click sur le fichier 1000d.prg, de sorte que le nom soit noirci. Ouvrir le menu Options et sélectionner la ligne Installer une application. On va vous demander le type de document. Entrer Z et cliquer sur OK. Ouvrir de nouveau le menu Options, cette fois pour sauvegarder le bureau. Pour lancer l'application, ouvrir la disquette et faire un double cliquage sur le nom du source Basic. Le programme 1000d se charge. Ensuite, selon le nom de l'application, une des actions suivantes est effectuée.

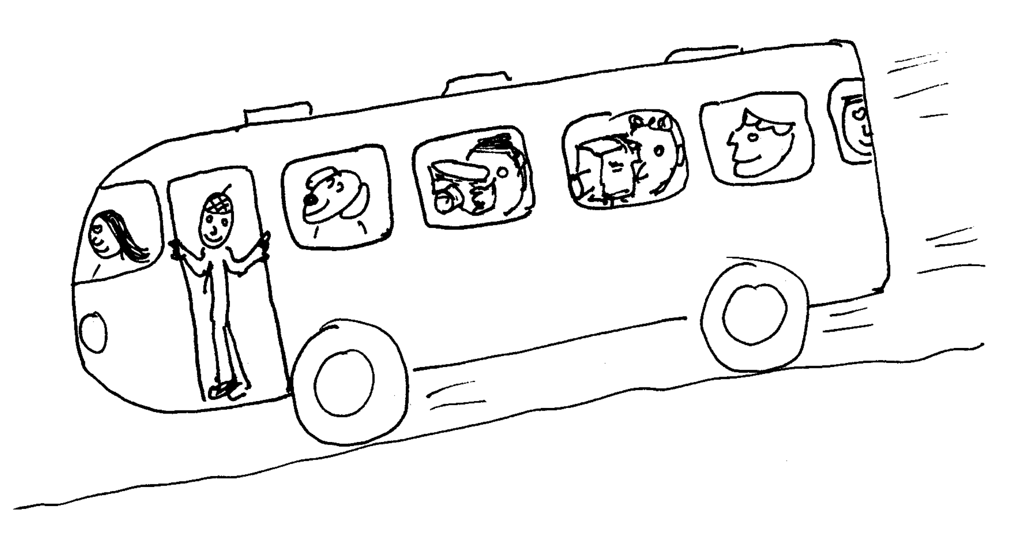
• Si le nom contient la suite de 4 lettres HELP, comme par exemple `thelp1.z`, le fichier est chargé dans la zone Help, et la fenêtre Help est ouverte. Pour les versions du TOS $\geq$ 1.4, le nom de l'application comprend le chemin des répertoires (par exemple `c:\help\1990\f12.z`).

• Si le nom contient MATH ou STND, le fichier est chargé dans la zone bibliothèque (exemples `math.z`, `stndx.z`).

• Sinon, le fichier est chargé dans la source et exécuté.

2

Visite rapide du Basic 1000d



Ce chapitre, qui a pour but de vous permettre d'éditer vos premiers programmes en Basic 1000d, présente les points les plus importants de l'éditeur.

Vous pouvez lancer le programme 1000D.PRG sous n'importe quelle résolution. Cependant, pour vos premiers essais, il vaut mieux utiliser la haute ou moyenne résolution car les titres se trouvent écrits en abrégé en basse résolution.

Notations

Dans les descriptions de syntaxe, les éléments placés entre crochets [ ] sont optionnels et les éléments placés entre accolades { } peuvent être répétés un nombre quelconque de fois.

Bouton

Désigne le bouton gauche ou droit de la souris.

Touche F

Désigne les touches de fonctions.

La notation [s] F3 désigne l'appui sur la touche de fonction F3 et Shift. De même, les notations [a] et [c] désignent respectivement l'appui sur Alternate et Control. Par exemple [c] Z désigne l'appui sur la touche Z avec Control enfoncé, [ca] ↑ , [ca] ↓ , [ca] ← et [ca] → désignent les appuis sur les flèches avec Control et Alternate enfoncés.

Nous noterons aussi F1 à F40 les touches avec et sans Shift et Alternate (c'est l'ordre des cases des damiers, Shift rajoute 10 et Alternate 20).

Break

A tout moment, sauf en cas d'opération système, on peut revenir à la fenêtre Basic en appuyant en même temps sur Control, Shift et Alternate, puis en relâchant ces touches. Ce retour provoque une vérification de la source qui peut prendre quelques secondes.

Lecture du mode d'emploi

En première lecture il est conseillé de sauter les paragraphes comme

Δ Les textes présentés ainsi sont plus difficiles ou font appel à des notions non encore exposées. A lire comme références ou si vous voulez devenir expert en Basic 1000d.

Certains des exemples demandent quelques minutes de calcul (calculer et écrire un nombre de 130100 chiffres . . .). D'autres apparaissent sous forme d'exercices, avec une solution en appendice.

Pour bien assimiler le langage, dès que vous saurez éditer des textes, essayez vos propres exemples, et exécutez les avec le débogueur.

Pour poursuivre l'étude du Basic 1000d, étudier ensuite les programmes des bibliothèques Stnd et Math. Pour une initiation au calcul formel, nous vous conseillons le livre de Davenport et al (1987).

Damier (ou menu) et fenêtre

Vous voyez apparaître, sur les premières lignes en haut de l'écran, un damier (ou menu). Le titre du damier, BASIC 1000D, est aussi le nom de la fenêtre ouverte. La fenêtre, sans bordures, occupe tout l'écran en dessous du damier.

Les diverses cases du damier correspondent à des commandes disponibles pour la fenêtre ouverte. Ces commandes peuvent être lancées soit en cliquant le bouton gauche sur la case, soit avec les touches de fonctions.

Cliquer sur la case FILES ou appuyer sur F2. Le menu FILES s'ouvre. Dans ce menu, seules les touches de fonctions et le bouton gauche sont actifs. Les autres touches font revenir au menu Basic. Sélectionner la case Dir, puis taper Return. Cela liste le répertoire du disque et retourne au menu Basic. Le contenu de la disquette reste écrit dans la fenêtre Basic.

La fenêtre DESK

Cliquer sur la case DESK ou appuyer sur F1 fait apparaître un bureau GEM à menus déroulants. Le premier menu déroulant permet de sélectionner les accessoires (les fichiers *.ACC). Le deuxième menu permet de revenir dans les fenêtres Basic ou Edit Source, ou de quitter le Basic.

Il est recommandé pour éviter les plantages, de refermer les accessoires avant de quitter cette fenêtre. De toutes façons, le Basic ne leur donne pas la main en dehors de la fenêtre DESK et sauf par programmation.

Δ Le fonctionnement de la fenêtre DESK est analogue à la partie du programme B_USER de la bibliothèque STND qui affiche le menu. L'étude de ce programme vous montrera que le Basic ouvre une fenêtre GEM, ce qui fait que les accessoires ouverts et non refermés, se trouvent désactivés sous cette fenêtre lors du retour dans la fenêtre DESK. Si un accessoire est bien programmé, un click sur son nom dans le menu réactivera alors l'accessoire.

Les autres fenêtres du Basic 1000d se présentent sous forme de damiers et non sous forme de menus déroulants. Les programmes en Basic peuvent appeler pour leur propre usage les menus déroulants (qui donnent accès aux accessoires) ou les damiers (qui se sélectionnent à la souris ou par les touches de fonctions).

La fenêtre Basic

La fenêtre Basic (nous abrégeons ainsi le titre Basic 1000d) qui s'ouvre au lancement du programme est celle du mode direct (exécution directe des commandes du Basic).

A l'aide des touches mouvements on peut se déplacer et écrire facilement sur tout l'écran sauf le damier. En attendant la liste complète des mouvements, nous vous laissons découvrir l'utilisation des flèches (↑ , ↓ , → et ←), de Home, Clr (c'est à dire [s] Home) Delete et [s] ↓ .

Cette écriture ne modifie que l'écran, tant que Return ou Enter n'est pas utilisé.

Taper ([CR] désigne la touche Return ou Enter)

```
PRINT $100 [CR]
```

L'instruction Basic est exécutée en mode direct : la fenêtre CALCUL s'ouvre, la valeur décimale du nombre hexadécimal 100 s'écrit, puis on retourne au menu Basic. On retrouve alors la fenêtre Basic initiale, avec en plus la sortie du `PRINT`.

Réutilisation de l'écran

Tout ce qui a été écrit sur l'écran au clavier ou par des instructions comme `print` peut être réutilisé. Par exemple, taper (cette fois utiliser [c] P pour écrire `print`)

```
print (1+x)^7 [CR]
```

puis remonter sur le 7 de `(1+x)^7` et le changer en 5 et taper [CR].

Comme autre exemple, exécuter l'instruction

```
print/c/"print 2+2"
```

puis taper [CR] après avoir déplacé le curseur sur la ligne écrite.

Cependant, une fois l'écran de la fenêtre Basic effacé (par la touche Clr par exemple), tout ce qui était écrit est perdu. Pour des calculs plus consistants, on n'utilisera pas le mode direct, mais on éditera un programme, puis on l'exécutera.

Source et édition de la source

La source est un ensemble de lignes (programme ou autre) en mémoire. Pour l'éditer, ouvrir la fenêtre Edit Source en cliquant la case F10 (EDIT).

Examinons comment entrer le programme suivant, qui permet d'apprécier la précision des calculs en itérant 20 racines carrées puis 20 carrés.

```
w=2
for i=1 to 20
  w=sqr(w)
next i
for i=1 to 20
  w=w^2
next i
print w
```

Sortie (300 ms)

```
0.2000000021~ E+1
```

Après avoir vidé la source si elle n'est pas vide (Case F13 New), taper d'abord la première ligne, puis sur Return ou sur la flèche ↓ . Taper de même les trois lignes suivantes. Noter que `next` s'obtient par [c] N et `for` par [c] FD (c'est à dire appuyer sur F puis D tout en maintenant la touche Control enfoncée, puis relâcher Control).

Cette première frappe effectuée, étudier l'effet des flèches. Noter que vous ne pouvez vous déplacer que sur les quatre lignes de la source et la ligne suivante. Toute modification effectuée sur l'écran, sera également effectuée dans la source.

Les lignes 5–7 du programme ressemblent aux lignes 2–4, on peut les éditer par des manipulations de bloc. Tout d'abord, définir le bloc, comme étant l'ensemble des lignes 2 à 4. Appuyer pour cela le bouton gauche sur la ligne 2, puis tout en le maintenant appuyé, déplacer la souris sur la ligne 4, ce qui dessine un rectangle en vidéo inverse, et relâcher le bouton. Le bloc est alors visualisé par des marques sur la colonne de gauche.

Remettre le curseur en fin de source, s'il n'y est pas, soit avec un click du bouton gauche (mais sans déplacement cette fois), soit avec les flèches. Répéter le bloc (case Recopy du menu BLOCK ou touche [ca] R). Remonter le curseur et changer `sqr(w)` en `w^2` dans la sixième ligne.

Pour ce changement, vous placez le curseur sur le `s` de `sqr(w)`, tapez `w^2`, et effacez la fin de ligne par [s] Delete.

Taper la dernière ligne.

Sauvegarde et chargement

Pour sauvegarder le programme que vous venez d'éditer, cliquer le bouton gauche sur Save (case F12) ou appuyer sur [s] F2. Vous pouvez alors soit taper le nom du fichier au clavier, suivi de Return, soit cliquer le bouton pour appeler le sélecteur de fichier du système. Si vous donnez le nom d'un fichier déjà sur disque, un avertissement vous permet d'éviter le recouvrement de l'ancien fichier.

Il est très désagréable de ne plus pouvoir relire un fichier important sur disque. Basic 1000d vous permet de vérifier que la sauvegarde s'est bien passée, en relisant le fichier et en comparant avec le fichier en mémoire. Pour cela cliquer Verify du menu FILES. Dans l'exemple proposé, comme la source tient en entier dans le tampon disque, la vérification a lieu dans la mémoire vive. Il faut recliquer sur Verify pour forcer une vraie relecture du disque. Cette vérification exige ici que l'extension du fichier soit Z.

La case Merge (F11) permet de lire un fichier et de le mettre à la fin de la source précédente. La commande Verify est également disponible en lecture.

Dans le menu FILES on trouvera d'autres opérations de chargements et sauvegardes (par exemple sauvegarde d'un bloc et chargement devant une ligne quelconque de la source).

Exécution du programme

Lancer le programme (case Run ou touche F8). La fenêtre RUN s'ouvre, et le programme y écrit ses réponses. Après exécution, la fenêtre RUN se transforme en fenêtre Basic, sans effacement des résultats.

Correction des erreurs

Que se passe-t-il en cas d'erreur ? Par exemple, supposons que la ligne 5 soit

```
for i=1 to 2O
```

où vous avez tapé la lettre O au lieu du chiffre 0 dans 20.

L'exécution de ce programme s'arrête avec le message

```
*ERREUR* INSTRUCTION ILLEGALE
for i=1 to 2?
5.for i=1 to 2O
```

La première ligne donne la cause de l'erreur. Le Basic à décodé l'instruction comme étant une boucle de 1 à 2, puis a trouvé la lettre O, qui n'est pas prévue dans la syntaxe de l'instruction `for`. D'où le diagnostic d'instruction illégale comme cause de l'erreur. La deuxième ligne indique, avec un point d'interrogation, l'endroit où l'erreur s'est produite. La troisième ligne, numérotée, recopie le texte entier de la ligne source en erreur.

Comment corriger une erreur ? Il faut d'abord en comprendre la cause. Pour vous aider dans la recherche des causes, ce manuel contient une description des messages d'erreurs, ainsi que des indications sur les causes possibles. De plus vous pouvez examiner le type des mots de l'instruction, (commande `type`) et le contenu des variables (par `print`).

Passons maintenant à la correction de la source. Plusieurs méthodes sont possibles.

Par la commande n.texte

Corriger l'erreur sur la ligne numérotée telle qu'elle est écrite, dans la fenêtre Basic, dans le message d'erreur. Il faut ensuite appuyer sur Return pour que la modification soit enregistrée dans la source.

En retournant dans la fenêtre Edit Source

La case EDIT, ou le cliquage du bouton droit sur la ligne numérotée, ouvre la fenêtre Edit Source, avec le curseur positionné sur la ligne en erreur. La correction de la ligne est alors enregistrée dans la source sans qu'il y ait besoin de valider par Return.

Débogage

Basic 1000d est doté d'un système de mise au point des programmes très perfectionné qui rend la détection d'erreurs logiques de programmation rapide et aisée. Comme exemple nous allons contrôler et vérifier l'exécution du programme que nous venons de vous faire éditer.

Si vous n'avez pas tapé le programme, comme les essais qui suivent facilitent l'assimilation du langage Basic 1000d, avant de poursuivre la lecture, reprenez cette notice de façon active en testant vous-mêmes dorénavant tous les exemples proposés.

Lancer le programme en sélectionnant la case DEBUG. La fenêtre DEBUG s'ouvre. Elle contient un damier de commandes, et quelques lignes source. Ce sont les lignes à partir de la ligne à exécuter, qui est pour le moment la première ligne. Ces lignes sont toujours numérotées à partir de 1, en vue de faciliter le débogage, le véritable numéro n de la première ligne est indiqué par le message

```
le prgr va exécuter la ligne n
```

Le débogueur est donc positionné sur la première instruction qui n'a pas encore été exécutée. Appuyez sur Return, la première instruction a été exécutée, le débogueur est positionné sur la deuxième ligne. Appuyez encore plusieurs fois sur Return (ou sur F1). Les instructions sont exécutées une à une, c'est le pas à pas.

Vous pouvez aussi cliquer, au cours de ce pas à pas, sur les divers noms écrits sur la page, avec le bouton gauche. Par exemple cliquez sur `print`, cela vous indique que `print` est une commande. Cliquez sur `w`, cela vous indique que `w` est de type var et écrit sa valeur.

Cliquez maintenant le bouton droit sur la ligne `for i=1 to 20`. Le programme s'exécute jusqu'à cette ligne. L'effet du cliquage a été de placer un point d'arrêt (breakpoint) et de lancer l'exécution.

Retournez à la fenêtre Basic par Arret (F2 ou touche A). Vous pouvez taper

```
print 1/w
```

pour examiner la valeur de 1/w.

Ouvrez la fenêtre Edit Source. Le curseur se trouve sur la ligne où était arrivé le débogueur. Modifier la source en rajoutant `print w`. Pour cela placer le curseur sur `next`, puis appuyer sur Insert, et taper la nouvelle ligne. La fin du programme est donc maintenant :

```
for i=1 to 20
  w=w^2
  print w
next i
print w
```

Sélectionner la case DEBUG+ ([a] F9), cela fait revenir le débogueur sur la ligne où vous l'aviez interrompu, et permet de poursuivre l'exécution du programme. Appuyer sur Return le débogueur se place sur la ligne

```
w=w^2
```

Cliquer Fois (Case F7 ou touche F), puis entrer 3 [CR] et [CR]. Cela lance l'exécution jusqu'à ce que le programme trouve trois fois la même ligne. Pour voir l'écran du programme appuyez sur E ou F6. Pour revenir au débogueur appuyez sur K ou F10.

Cliquer Cycl (Case F8 ou touche H). Cela lance l'exécution jusqu'à ce que le programme revienne sur la même ligne.

Cliquer Loop xit (Case F20 ou touche L). Cela lance l'exécution jusqu'à la sortie de la boucle.

Parmi les fonctions de débogage non présentées ici, notons la case Rts qui permet de revenir d'un sous-programme, et la possibilité de programmer, en Basic 1000d, le débogueur (par exemple pour imprimer certaines variables dans la fenêtre DEBUG ou pour exécuter jusqu'à ce qu'une condition devienne vraie).

Bibliothèques

En outre de la source, Basic 1000d dispose d'une deuxième zone mémoire pouvant contenir des programmes en Basic 1000d, c'est la bibliothèque (Library).

Chargez la bibliothèque Math, en sélectionnant la case Merge,l ou Load,l du menu FILES ou LBR/HLP, puis spécifier le fichier math.z.

Ouvrir la fenêtre Edit Source, la bibliothèque est invisible. Pourtant, tous ses programmes sont utilisables. Essayons le programme B_USER. Ce programme pourrait s'appeler en mode direct par

```
B_USER
```

qui est identique à

```
gosub B_USER
```

Pour appeler ce programme particulier il est inutile de taper ces commandes, il suffit de cliquer la case B_USER.

Vous voyez apparaître un damier, mais ce damier est géré par le programme B_USER. Il vous permet d'appeler sans effort d'autres programmes de la bibliothèque MATH.

Sélectionnez par exemple la case Intègre et entrez `x/(x^2+1)`. Immédiatement vous obtenez une valeur de $\int \frac{x}{x^2+1} dx$. Appelez de nouveau B_USER et cliquer la case Système pour lui faire résoudre, de façon exacte, le système d'équations :

$$\begin{cases} x^5 + y^5 = 33 \\ x^2 + y^2 = 5 \end{cases}$$

Le programme vous guidera pour l'entrée du système.

Maintenant sélectionnez la case L->S du menu LBR/HLP. Cela a pour effet de faire passer la bibliothèque dans la source, comme vous pouvez le vérifier en ouvrant la fenêtre Edit Source.

Cliquer alors la case T level (case F20) ce qui liste seulement les lignes avec des labels. Recherchez le label B_USER, et cliquez le bouton droit sur sa ligne. Vous obtenez alors le listing du programme, et dans quelques jours vous serez à même de pouvoir le modifier à votre gré, ou d'écrire vos propres programmes B_USER.

Modes d'emploi (Help)

Il y a dans le Basic 1000d trois zones contenant du texte. Nous avons vu les zones Source et Bibliothèque.

La troisième zone, Help, permet la création de modes d'emploi ou d'aide-mémoire.

Appuyer sur la touche Help, puis sur CR. Cela charge le fichier Help.z de démonstration et ouvre la fenêtre Help. Appuyer de nouveau sur Help pour commencer la lecture du mode d'emploi, puis sur la touche ↓ pour la continuer.

Ce fichier contient des programmes en exemples, qu'il est possible de faire passer dans la source par appui sur [s] F10. Avant d'effectuer de tels transferts, revenez au menu Basic (case F10) et videz la source (New case F13). La touche Help rouvre maintenant la fenêtre Help au même point.

3

La fenêtre Edit Source



Rôle

La fenêtre Edit Source permet de créer et de modifier la source (les programmes en Basic).

Source et options d'affichage

Source

C'est un ensemble de lignes en mémoire représentant un programme.

∆ La source est un fichier Z, c'est à dire un fichier ASCII où la séparation des lignes est codée par `chr$`(0), et non par `chr$`(13) et `chr$`(10). De plus, il y a quelques restrictions sur le texte des lignes éditables. Les caractères `chr$`(13), `chr$`(25) et `chr$`(26) ne peuvent être édités. Les espaces sont supprimés en début et fin de ligne, devant “:” et “"”.

Mis à part quelques restrictions, il est possible d'insérer un texte quelconque. Cela présente l'avantage de pouvoir utiliser l'éditeur pour des textes autres que du Basic, comme par exemple des modes d'emploi utilisables par Help. Certains préfèrent avoir une vérification de la syntaxe au moment de l'entrée de ligne. C'est que souvent la correction d'erreurs est malaisée dans de nombreux langages de programmation. Ce n'est pas le cas en Basic 1000d. La correction des erreurs au cours de l'exécution est extrêmement simple. En effet la ligne en erreur est présentée pour permettre une correction immédiate, et on peut ensuite, après correction, faire repartir le programme du point où il avait été arrêté.

Chaque ligne de l'écran correspond à une ligne de la source. Lorsqu'une ligne est trop longue pour être écrite complètement une flèche apparaît sur la dernière colonne de la ligne.

La longueur de ligne maximum permise est environ 3600 caractères.

L et T level (F20)

Le listing de la source présente usuellement des lignes consécutives de la source. Un autre type de listing écrit uniquement les lignes qui portent un label, ce qui permet de retrouver rapidement les sous-programmes.

On passe d'un type de listing à l'autre en cliquant sur la case F20 (niveau <T>out ou <L>abel).

On revient aussi au niveau T en cliquant le bouton droit, ou en appuyant sur Return.

Indent (F17)

Le listing est indenté ou non, au choix, le changement s'effectuant en sélectionnant la case F17. En cas de Noindent, l'affichage est nettement plus nerveux.

En cas d'indentation, il y a décalage, à la manière du Pascal, des structures et boucles. Si la source est longue, aller du début en fin de source peut prendre quelques secondes. En effet, l'indentation n'est pas conservée en mémoire, mais recalculée avant affichage.

No menu (F16)

L'appui sur [s] F6 supprime ou remet le menu.

Résolution ([s] Tab)

L'appui sur [s] Tab change la résolution. Dans le cas d'un écran monochrome, avec No menu on peut représenter jusqu'à 50 lignes de la source sur l'écran.

Numéro de ligne

Les lignes sont numérotées 1, 2, 3, ..., mais seul le numéro des lignes M et D est affiché. C'est l'éditeur qui se charge de la numérotation.

Ligne M

C'est la ligne du curseur. Sa valeur est écrite sur les cases F27 et F28 du damier.

Les cases correspondantes sont actives et permettent de modifier la valeur de M. La syntaxe pour entrer la ligne, qui ne se limite pas seulement aux numéros ou aux labels, sera étudiée avec la commande `#ligne` de la fenêtre Basic.

Ligne D

La dernière ligne de la source. Le numéro D, qui est donc aussi le nombre de lignes de la source, est affiché sur les cases F25 et F26 du damier (cases non actives).

Espace mémoire libre

Le nombre k de kilo-octets encore utilisables par la source est écrit sur les cases F23 et F24 du damier.

Nombre de lignes modifiées

Le nombre de lignes de la source qui ont été modifiées depuis la dernière sauvegarde est écrit sur les cases F21 et F22 du damier.

Utilisation de la souris

Cliquer le bouton gauche de la souris sur le damier, ou utiliser les touches F1 à F40 pour les actions du menu.

Cliquer le bouton droit pour lister à partir de la ligne

Cliquer le bouton gauche dans la page pour déplacer le curseur.

Déplacer la souris bouton gauche appuyé pour définir le bloc.

Ecriture

Mode d'écriture (Insert ou Overwr, F30 et [s] Ins)

Ce mode est lisible (et modifiable) sur la case F30 du damier. Il est également donné par la fréquence de clignotement du curseur, qui est plus rapide en mode Overwr. Il est aussi modifiable par appui sur [s] Insert. Il conditionne la façon d'écrire sur l'écran, soit par insertion sous le curseur ou par écriture sur le curseur.

Caractères exotiques

On peut écrire le symbole "ô", de code Atari $93=147 en tapant 147 ou H93 (chiffres du clavier numérique) tout en maintenant Alternate enfoncé. Cette méthode est acceptée pour tous les codes Atari sauf 0. D'autres codes s'obtiennent par Alternate + lettre.

La table de ces codes Atari est donnée par la commande ASCII (F4) du menu HELPS (F3).

Noter que l'insertion dans la source des codes 25, 26 (qui contrôlent le début et la fin de la source) et 13 est interdite, et celle des codes 1 à 10 est très déconseillée. Les codes 1 à 10 présentent l'inconvénient de ne pouvoir être recherchés dans la source (Voir FND/CHG).

Mots clefs

L'appui sur des lettres de A à Z tout en maintenant Control enfoncé écrit, lors du relâchement de Control, un mot clef du Basic. Si on appuie ensuite sur Control et "+" du pavé numérique, ce que nous notons [c] +, ce mot clef est remplacé par le mot clef suivant du Basic. On peut rappuyer sur [c] + (ou maintenir ces touches enfoncées) pour faire défiler la totalité des mots clefs du Basic.

Par exemple la frappe de [c] P donne `print`, la frappe de [c] PA ou de [c] P suivi de [c] + donne `precision`, la frappe de [c] PB ou de [c] PAA ou de [c] P suivi de deux frappes sur [c] + donne `procedure`, etc.

On dispose ainsi d'abréviations d'une, deux ou trois frappes pour tous les mots clefs.

Δ Soit n le numéro d'ordre du mot clef écrit par [c] x (x désignant n'importe quelle lettre). La frappe de [c] x y z ... écrit le mot clef numéro n + d(y) + d(z) + ... où d(A)=1, d(B)=2, ..., d(Z)=26.

Les touches mouvements

La commande Editing du menu HELPS donne le résumé des mouvements possibles. CR désigne la touche Return ou Enter.

↑ et ↓

Déplacent le curseur verticalement, ou font défiler la fenêtre.

← et →

Déplacent le curseur horizontalement, et font défiler la ligne.

[s] ↑ ou [c] Home

Curseur sur la première ligne de la source

[s] ↓

Curseur après la dernière ligne de la source

[s] ← et [s] →

Curseur sur le premier ou dernier caractère de la ligne.

[c] ↑ et [c] ↓

Page de la source précédente ou suivante.

[c] ← et [c] →

Déplacent le curseur par mots.

Tab et [c] Tab

Curseur sur la position de tabulation suivante ou précédente. Les positions de tabulation sont espacées de 8 caractères.

Home et [s] CR

Curseur au coin gauche en haut ou en bas de la fenêtre.

Delete et Backspace

Effacent le caractère sous le curseur ou devant le curseur.

[s] Delete et [s] Backspace

Effacent la ligne après le curseur ou devant le curseur. De plus, les caractères effacés sont mis dans le tampon d'effacement.

[c] Delete

Efface la ligne et la met dans le tampon d'effacement. Le curseur se retrouve sur la ligne qui suivait.

Clr

Vide la ligne et la met dans le tampon d'effacement. Supposons que Clr a été effectué, sans que Ins ait été appuyé. Si on appuie ensuite sur CR, la ligne est effacée (Clr suivi de CR équivaut à [c] Delete). Si on quitte la ligne autrement, dans la source il reste une ligne vide.

Undo

Ecrit le tampon d'effacement. Cette commande permet de recopier rapidement une ligne ou portion de ligne. Pour cela on met d'abord la (portion de) ligne dans le tampon d'effacement en utilisant une des commandes [s] Delete, [s] Backspace ou Clr, immédiatement suivie de Undo pour rétablir la ligne. Mais attention, comme les espaces en début et fin sont ignorés par Undo, il se peut qu'il manque des espaces dans la ligne récrite. Ensuite, on peut recopier le texte du tampon d'effacement autant de fois que nécessaire, à n'importe quel point de la source, et même dans d'autres fenêtres.

[s] Undo

Ignore les dernières modifications effectuées dans la ligne M. Il s'agit de toutes les modifications depuis que M a pris sa valeur présente.

Δ Edit Source est un éditeur ligne, qui enregistre dans la source les modifications de la ligne M lorsqu'une commande lui fait quitter cette ligne. [s] Undo lui fait retracer la fenêtre sans modifier la source. Un Break effectué dans la fenêtre Edit Source se comporte de façon analogue, provoquant un retour dans la fenêtre Basic sans enregistrer les dernières modifications.

CR

Déplace le curseur sur la ligne suivante. Lorsqu'on tape CR sur une ligne vide de la source, cette ligne est effacée de la source.

Ins

Insère une ligne vide devant la ligne M. On peut alors taper le texte d'une ligne, puis CR. Une nouvelle ligne vide apparaît, ce qui permet de continuer l'insertion de lignes. On peut insérer des lignes vides, en tapant CR.

La ligne vide d'insertion n'est pas conservée dans la source, si on en sort autrement que par CR.

Help

Ouvre la fenêtre Help

Esc ou case BASIC (F10)

Ouvre la fenêtre Basic

[s] **Esc**

Ouvre la fenêtre Basic et y écrit le texte de la ligne M. Cela peut permettre d'exécuter en mode direct une instruction qui était écrite dans la source, ou d'afficher la totalité d'une ligne de plus de 80 caractères.

Bloc

Le bloc est un ensemble de lignes consécutives de la source. Il peut être déplacé, copié, effacé, sauvegardé, et imprimé. Le bloc est visualisé par des marques dans la colonne de gauche.

Les commandes principales relatives au bloc sont obtenues au clavier par appui simultané sur Control, Alternate et une autre touche.

Nous avons déjà vu comment on peut définir le bloc, s'il est entièrement sur une page, avec le bouton gauche appuyé. Si le bloc est plus grand, définir son début et sa fin séparément.

Ligne A

La première ligne du bloc. La ligne M (du curseur) est prise pour ligne A par la commande Mark a du menu BLOCK ou par appui sur [ca] ↑ .

Inversement, on peut déplacer le curseur sur la ligne A par la commande Goto a du menu BLOCK ou en appuyant sur [ca] ← .

Ligne B

La dernière ligne du bloc. La ligne M (du curseur) est prise pour ligne B par la commande Mark b du menu BLOCK ou par appui sur [ca] ↓ .

Inversement, on peut déplacer le curseur sur la ligne B par la commande Goto b du menu BLOCK ou en appuyant sur [ca] → .

Delete du menu BLOCK ou [ca] D

Efface le bloc, après avertissement.

Move du menu BLOCK ou [ca] M

Déplace le bloc devant la ligne M, et met M après le bloc. Le bloc reste défini sur les mêmes lignes.

Recopy du menu BLOCK ou [ca] R

Recopie le bloc devant la ligne M, et met M après la copie. Le bloc d'origine reste défini et inchangé.

Δ Cela n'est plus vrai si M=B+1, le bloc doublant de volume, en accord avec la règle que toute ligne insérée entre A et B+1 est absorbée par le bloc. La raison est que le pointeur de fin de bloc est en réalité le début de la ligne B+1, et que l'insertion augmente ce pointeur.

Imprimer le bloc

L'impression (Print,b) est étudiée avec le menu PRINTER.

Sauvegarder le bloc

La sauvegarde (Save,b) est étudiée avec le menu FILES.

Cacher le bloc

Si vous ne pouvez plus supporter la présence d'un bloc, vous pouvez le cacher en redéfinissant la ligne A en dessous de B.

Marques

On dispose d'une pile permettant de mémoriser 4 positions (marques) dans la source. C'est la ligne M qui est mémorisée. Lors du rappel, la ligne mémorisée est placée en haut de l'écran. En plus de ces quatre marques, la dernière ligne modifiée de la source est automatiquement gardée en mémoire.

Les commandes relatives aux marques sont obtenues au clavier par appui simultané sur Control, Shift et une autre touche.

Push pg du menu BLOCK ou [cs] ↓

Met sur la pile la ligne M. La pile s'enfonce et la marque du bas de la pile est alors perdue.

Pop pg du menu BLOCK ou [cs] ↑

Met le curseur sur la ligne mémorisée en haut de la pile. La pile est alors permutée de façon cyclique, le haut de la pile se retrouvant en bas de la pile.

Swap pg du menu BLOCK ou [cs] Help

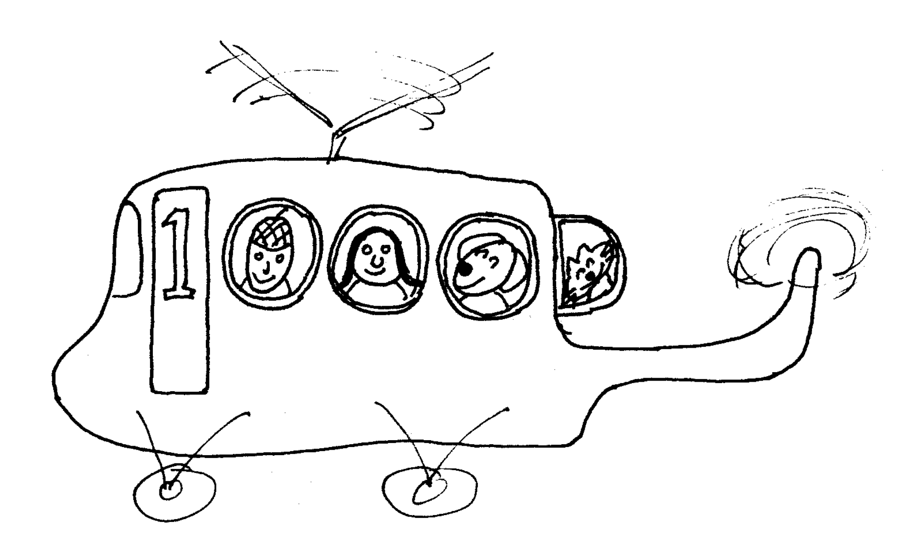
Met le curseur sur la ligne mémorisée en haut de la pile, et met l'ancienne ligne M à la place de cette ligne sur la pile. Les autres marques de la pile ne sont pas modifiées.

Last mod du menu BLOCK ou [cs] Undo

Met le curseur sur la ligne de la source qui a été modifiée en dernier. Cette marque est indépendante de la pile des quatre marques.

4

La fenêtre Basic 1000d



Rôle

Le menu Basic 1000d est le menu de base qui s'ouvre au chargement du logiciel. Il permet d'ouvrir les autres fenêtres et d'exécuter des instructions du Basic en mode direct. C'est aussi l'état d'attente dans lequel on revient après exécution d'un programme ou après Break.

Dans le chapitre d'introduction, nous vous avons présenté la commande Dir du menu FILES. Cette commande, qui a d'abord nécessité l'ouverture du menu FILES, revient dans la fenêtre Basic pour y écrire ses résultats que l'on peut consulter tranquillement. La plupart des commandes de l'éditeur se comportent de la même manière, avec retour dans la fenêtre Basic.

On ne peut pas accepter que des commandes comme Dir écrivent leurs résultats directement dans la fenêtre Edit Source, qui est un reflet fidèle de la source. Lorsqu'une telle commande y est effectuée, il y a également passage dans la fenêtre Basic, qui sert à attendre que les résultats aient été examinés avant de permettre le retour dans la fenêtre Edit Source.

Ecran

L'écran de la fenêtre Basic diffère de celui de la fenêtre Edit Source. Il n'y a pas de défilement, ni horizontal ni vertical et les commandes peuvent s'écrire sur plusieurs lignes.

L'écran de toute fenêtre qui attend une entrée clavier (par exemple dans l'instruction `input` du Basic) est similaire à cet écran, sauf pour les fenêtres Edit Source et Help.

Ligne texte

Les caractères s'écrivent sur l'écran suivant 25 (ou 50 si `resolution`=3) lignes horizontales que nous appelons lignes écran. L'écriture d'une commande ou d'une instruction peut demander plus de caractères qu'il n'y en a dans une ligne écran (80 ou 40 si `resolution`=0). Un tel long texte est entièrement écrit sur l'écran, mais sur plusieurs lignes écran, qui forment ce que nous appelons une ligne texte.

Après l'appui sur Clr il y a autant de lignes textes que de lignes écran. Lorsqu'on écrit en passant à la ligne suivante, on crée une ligne texte formée de 2 lignes écran. Il faut vraiment écrire, passer avec le curseur ne suffit pas. En

appuyant sur [s] ← ou [s] → , qui déplacent le curseur sur le premier ou dernier caractère de la ligne texte, on peut vérifier la longueur de la ligne texte. La ligne texte peut occuper tout l'écran.

Examiner aussi comment se rallonge la ligne texte, en mode Insert, lorsqu'on rajoute des caractères en son milieu. Noter que les effacements agissent sur la ligne texte du curseur.

Après appui sur Return ou Enter, c'est la ligne texte du curseur qui est validée. Si cette ligne était la dernière de la page, la première ligne texte de la page disparaît à jamais et l'écran remonte laissant de la place pour une nouvelle ligne. La longueur de la ligne texte, lors de la validation par Return, est limitée à 1790 caractères.

Réutilisation de l'écran

C'est une propriété importante de l'écran qui est décrite dans le chapitre d'introduction.

Heure et Date

On peut modifier leur valeur en sélectionnant une des cases F38 à F40. Ces cases sont également actives dans la fenêtre Edit Source, mais inactives dans les autres fenêtres. Il est possible d'effacer et remettre la date et l'heure par des commandes du Basic.

Exemple

Pour les effacer, exécuter les commandes :

```
nodate
noclock
```

puis pour remettre l'heure :

```
clock
```

Ouverture de la fenêtre Edit Source

En outre de la case EDIT (F10), qui ouvre sur la ligne M, on dispose d'autres facilités.

Bouton droit

Cliquer le bouton droit sur une ligne numérotée pour ouvrir la fenêtre Edit Source sur la ligne cliquée. Des lignes numérotées sont écrites dans la fenêtre Basic par de nombreuses commandes. Nous avons déjà vu l'exemple d'erreurs de programme. Comme autres exemples, citons les commandes de recherche, et le listing du débogueur.

Δ Cette commande utilise seulement le numéro de la ligne. Cela permet par exemple d'ouvrir sur la ligne 107 en écrivant en début de ligne "`107.`" puis en cliquant le bouton droit sur cette ligne. Mais attention de ne pas appuyer sur [CR] après avoir tapé "`107.`". Cela viderait la ligne 107. La commande est valide avec les numéros relatifs (1≡) du débogueur, après retour dans la fenêtre Basic.

Bouton gauche

Déplacer la souris bouton gauche appuyé. Cela remplit le tampon d'effacement avec le texte des lignes sélectionnées et ouvre la fenêtre Edit Source sur la ligne M. Cette commande permet d'écrire dans la source des données sorties par un programme. On les recopie ensuite par appui sur Undo n'importe où dans la source.

New

La commande New (case F13) vide l'ancienne source avant d'ouvrir la fenêtre Edit Source.

Commande #ligne

Cette commande ouvre la fenêtre Edit Source sur la ligne spécifiée. On peut aussi l'appeler par la case EDIT... (F20) du menu Basic, et aussi dans la fenêtre Edit Source en cliquant sur M= (cases F27 et F28). Il faut alors rentrer `ligne` sans le symbole `#`. Voici quelques exemples :

`#ligne`	Ouvre
`#`	en tête de la source
`#D`	en fin de la source
`#12`	sur la ligne 12
`#gamma`	sur la ligne gamma

Les définitions suivantes permettent de définir la syntaxe de `ligne` dans la commande `#ligne`.

label

désigne un nom alphanumérique apparaissant une fois en tête de ligne suivi de ":". Par exemple, on peut écrire `GammA` ou `gamma` pour désigner la ligne

```
gamma:function(x)
```

Les majuscules sont ici équivalentes aux minuscules (règle différente de celle des labels lors de l'exécution).

n

désigne un entier décimal

sligne

désigne les façons suivantes de définir une ligne :

```
  M       ; Ligne du pointeur d'insertion
| D       ; Dernière ligne de la source
| A       ; Pointeur A (1ère ligne du bloc)
| B       ; Pointeur B (dernière ligne du bloc)
| label   ; La ligne du label
```

ligne

désigne les façons suivantes de définir une ligne :

```
  (vide)     ; Ligne 1
| n          ; Ligne n
| sligne
| sligne+n   ; sligne décalé de n
| sligne-n
```

Edition

Les diverses touches agissent de façon similaire dans les écrans Edit Source et Basic. Nous nous contentons ici de noter les différences entre les deux.

Les caractères 13, 25 et 26 peuvent être écrits, mais 0 reste interdit.

Les touches [s] ↑ , [c] ↑ et [c] Home sont identiques à Home.

Les touches [s] ↓ et [c] ↓ sont identiques à [s] CR (curseur en bas de page).

Clr efface tout l'écran, pas seulement une ligne.

CR valide la ligne texte.

Les touches suivantes n'ont aucun effet : [s] Undo, Esc, [s] Esc, et les commandes de marque et de bloc.

Le tampon d'effacement est commun à toutes les fenêtres (Basic, Edit Source, Help, . . .). Il se remplit lors des effacements et se recopie par Undo dans toutes les fenêtres, ce qui permet facilement le passage de données d'une fenêtre à l'autre.

Commande n.texte

Cette commande permet de modifier une ligne de la source sans ouvrir la fenêtre Edit Source.

Elle remplace la ligne n (où n est un nombre décimal) par `texte`, et déplace le pointeur d'insertion sur la ligne suivante (M $= n + 1$). Le nouveau `texte` pris pour la nouvelle ligne ne contient ni les espaces en tête ni ceux avant le premier "”" et ":" non entre guillemets.

A la différence de l'édition dans Edit Source, le premier caractère de `texte` doit être A–Z, a–z, "”", ":", "\" ou "@".

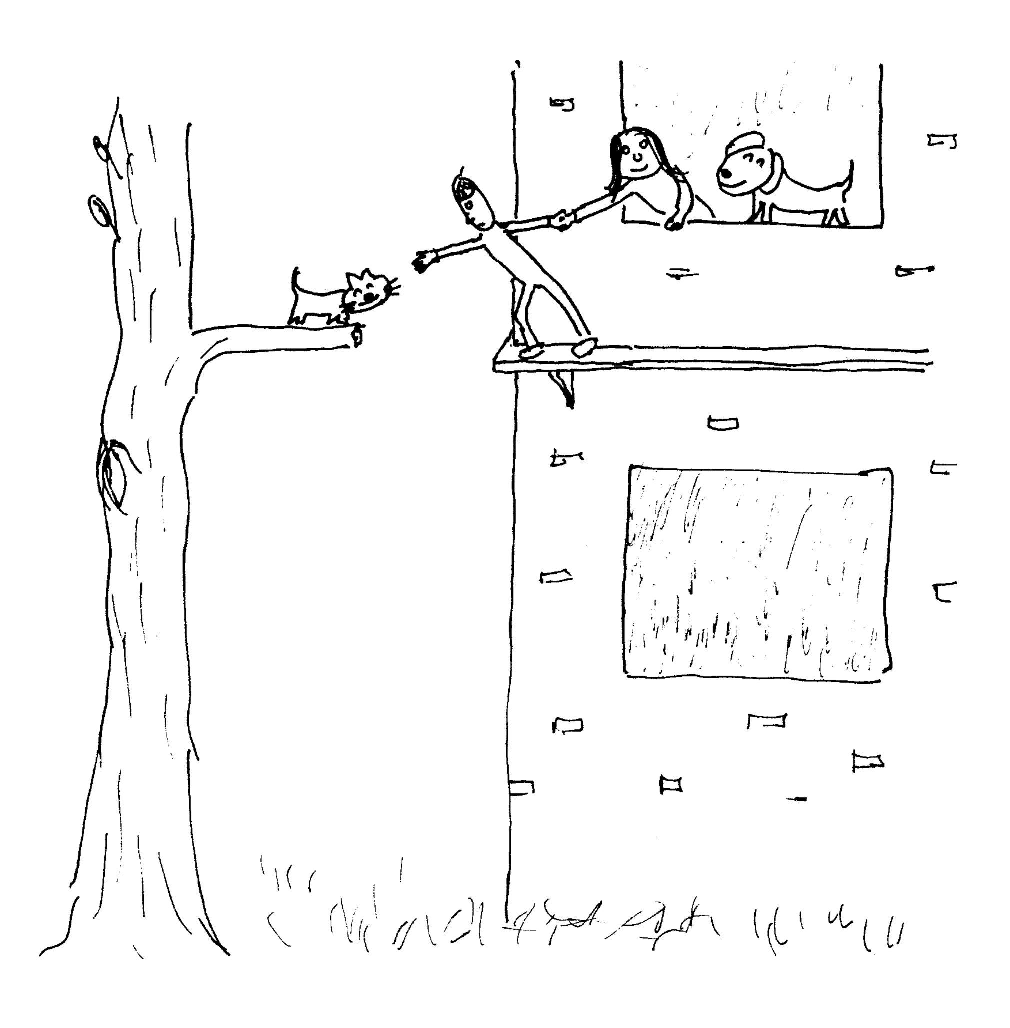
Si `texte` est vide, la ligne est remplacée par une ligne vide.

Cette commande est très commode pour la correction des erreurs. En effet, après détection d'une erreur, la ligne ayant causé l'erreur est écrite sous la forme `n.texte`. Les commandes de recherche et changement, elles aussi listent les lignes sous cette forme.

Pour corriger la ligne du programme en erreur, il suffit de déplacer le curseur sur le texte, de corriger et d'appuyer sur Return pour valider la correction.

5

La fenêtre Help



Les modes d'emploi HELP

L'entrée dans le menu Help se fait en appuyant sur la touche Help ou à partir du menu HELPS ou LBR/HLP, case Help. Le fichier chargé après l'appui sur la touche Help a nécessairement le nom HELP.Z, mais il est possible de charger dans Help des fichiers ayant un autre nom (voir les commandes disques du menu FILES). Pour lire la fenêtre Help, vous disposez d'une table des matières et d'un index extrêmement perfectionnés. Le fichier Help peut contenir des exemples de programmes en Basic. Ces exemples peuvent facilement être copiés dans la source.

Organisation du fichier Help

Le fichier Help contient des titres, rangés suivant leur niveau de A à E, et du texte ordinaire (niveau F). Lorsqu'on liste Help à partir d'un point donné, on peut choisir un niveau de A à F. Seuls les éléments de niveau supérieur ou égal sont présentés. De plus les titres et sous titres qui contiennent le point de départ de la page sont toujours présentés.

Autrement dit, un fichier Help a une structure d'arbre de profondeur maximum 6 et les commandes de la fenêtre Help permettent de se déplacer rapidement dans cet arbre. Par exemple, la lecture au niveau B ne retient que les textes (titres) jusqu'à la profondeur 2, et cliquer le bouton droit sur une ligne ouvre l'arbre jusqu'à la profondeur suivante.

Index

Entrer la CLEF, c'est à dire le mot sur lequel vous voulez des éclaircissements. On peut entrer la clef au clavier ([c] $\left\{\text{lettre}\right\}$ et [c] + pour le défilement donnent les mots clefs du Basic), qui est validée par simple écriture sans qu'il soit nécessaire d'appuyer sur Return, ou bien prendre un mot déjà écrit en plaçant la souris sur le mot et en cliquant le bouton gauche.

Ensuite chercher la clef d'abord dans les titres, puis dans le texte si nécessaire (par FIND du menu).

Clef

La clef est un nom contenant seulement des lettres non accentuées, des chiffres ou les symboles !#$%.?_âçéèêîô et dans lequel les majuscules ont la même valeur que les minuscules.

Lecture de deux pages

Lors de la lecture vous pouvez buter sur un terme expliqué à une autre page du mode d'emploi. Cliquer le bouton gauche sur ce terme, pour le mettre dans la clef, appuyer sur SWAP pour mémoriser la page en cours et aller chercher l'explication du mot par FIND.

Vous revenez à la page initiale par SWAP, et en rappuyant sur SWAP vous pouvez alterner entre les deux pages.

Test des exemples proposés

Vider d'abord la source (cliquer sur New), avant d'entrer dans la fenêtre Help. Cliquer sur P->S, ou sur [s] Esc qui a pour effet de copier l'exemple dans la source, et de vous renvoyer à la fenêtre Basic. Lancer ensuite l'exécution de l'exemple en cliquant le bouton gauche sur Run. Appuyer sur la touche Help pour reprendre la lecture du manuel.

Actions possibles

Cliquer le bouton gauche sur le damier, ou utiliser les touches de fonctions F1 à F20 (avec et sans Shift) ou des touches mouvements pour les actions du menu.

Cliquer le bouton droit sur une ligne du texte pour lister à partir de cette ligne (à un niveau inférieur si c'est un titre).

Cliquer le bouton gauche sur un mot du texte, ou l'écrire au clavier pour changer la clef.

Menu

A LEVEL

Liste seulement le sommaire, ce sont les titres de niveau A.

B LEVEL ... F LEVEL

Listent à partir du même point, à des niveaux différents.

PG UP

PG DOWN

Les cases Pg up (F5) et Pg down (F6) listent plus haut ou plus bas, sans modifier le niveau.

FIND

AGAIN

Ces actions utilisent la clef.

La case Find T (F7) cherche à partir du début la première occurrence du mot clef dans un titre. La case Again T (F17) cherche l'occurrence suivante dans les titres.

Les cases Find 1st (F8) et Again (F18) ont des effets analogues dans tout le texte.

En cas de long programme, il est possible que Find/Again liste le début du programme sans que le mot clef apparaisse. Il suffit de rappuyer sur Again (peut-être plusieurs fois) pour le faire apparaître.

Lorsqu'il n'y a plus de mot clef, la fin de Help est visible.

SWAP (F19)

Mémorise la page, et récrit l'ancienne page mémorisée.

PG IDEM (F9)

Remet la même page (utile après changement de la clef).

BASIC (F10)

Quitte le menu Help et revient à la fenêtre Basic.

P->S (F20)

Recopie le premier programme écrit sur la page en fin de Source et fait revenir à la fenêtre Basic. Pour recopier le deuxième programme écrit sur la page, il faut d'abord récrire la page avec ce programme en première position (cliquer le bouton droit sur le programme). On peut alors soit exécuter le programme, soit l'insérer dans ses propres programmes.

Clavier

L'écriture, possible seulement dans la dernière ligne, s'effectue comme dans la fenêtre Edit Source. Cependant, les touches qui dans Edit Source faisaient changer de ligne ont maintenant un rôle différent.

↑ ou [c] ↑

Comme Pg up

↓ ou [c] ↓

Comme Pg down

Home

Comme A level

[s] ↑

Liste en élevant le niveau (e.g. de C level on passe à B level)

[s] ↓

Liste en abaissant le niveau

Help

Liste le début du fichier Help, au niveau F

Esc

Retour à la fenêtre Basic (comme F10)

[s] Esc

Comme P->S (case F20)

[cs] Help

Comme Swap

Menu LBR/HLP

L'édition d'un programme, et sa mise au point n'est possible que dans la fenêtre Source. Le menu LBR/HLP contient les commandes permettant l'édition des zones Help et Bibliothèque, en particulier le transfert de ou vers la zone Source, sans passer par les disques. Les commandes disques du menu sont étudiées avec le menu FILES. Pour effectuer les transferts entre Help et Source, il est nécessaire que la Bibliothèque soit vide.

S->L (F1)

Transfère la source dans la bibliothèque, à la suite de l'ancienne.

,M->L (F2)

Transfère les lignes 1 à $M-1$ de la source dans la bibliothèque, à la suite de l'ancienne. Rappelons que M désigne la ligne du curseur.

L->S (F3)

Transfère toute la bibliothèque en tête de la source.

New,L (F4)

Efface la bibliothèque.

S->H (F11)

Transfère la source dans Help, à la suite de l'ancien.

,M->H (F12)

Transfère les lignes 1 à $M - 1$ de la source dans Help, à la suite de l'ancien.

H->S (F13)

Transfère tout Help en tête de la source.

Nohelp (F14)

Efface Help.

Edition de textes ASCII

On peut éditer des textes ASCII dans la fenêtre Edit Source, de préférence avec l'option Noindent. Il faudra tenir compte du fait que devant ":" et """ les espaces sont supprimés par l'éditeur.

Edition de ce manuel

La méthode suivante a été utilisée pour éditer le manuel que vous êtes en train de lire. Un exemple de programme est testé en l'écrivant en tête de source, suivi des instructions :

```
print/d/"Sortie (";justl$(mtimer);" ms)"
end
```

pour mesurer le temps d'exécution. La commande `end` sépare le programme testé du manuel en cours de rédaction. Les sorties du programme sont intégrées dans la source, par saisie avec le bouton gauche de la souris. Les marques (commandes avec [cs]) permettent de retrouver la page en cours.

Il était nécessaire de consulter d'autres textes (comme les fichiers des messages d'erreur du programme de publication TEX). Ces fichiers étaient placés dans la zone Help du Basic, permettant leur lecture en même temps que la rédaction du manuel.

Edition de modes d'emploi Help

Les instructions pour Help sont données en tête de ligne par le symbole "\" suivi d'une lettre (A–E, P, T, ou Z) écrite en majuscule.

Les titres doivent être écrits sur une seule ligne et précédés de \A ou \B ou \C ou \D ou \E suivant le niveau voulu. Le texte ordinaire se rentre sur des lignes de longueur quelconque. La fin d'une ligne s'affiche comme un espace. Un nouveau paragraphe s'indique par une ligne vide. Après \T un nouveau paragraphe commence sans indentation. Les espaces sont actifs après \T, pour permettre l'écriture de tables. Les programmes doivent être écrits comme en Basic, et précédés de \P. La fin du programme est indiquée par une ligne vide ou une ligne commençant par \T ou une autre commande de Help. Les paragraphes marqués Z commencent par \Z. Le fichier HELP.Z doit commencer par au moins une ligne arbitraire qui sera ignorée. Tout ce qui précède le premier titre \A n'est pas sorti par Help. On peut passer de la Source dans Help et inversement, sans opérations disque. Voir menu LBR/HLP

6

Autres menus de l'éditeur



Organisation des disques

Un nom de fichier complet, par exemple :

`A:\REP.EXX\BLEU.EYY\COL.Z`

se décompose de la façon suivante :

```
A:          Lecteur de disques
REP.EXX     Répertoire
BLEU.EYY    Répertoire
COL.Z       Fichier
```

Les noms des répertoires et fichiers y sont séparés par le symbole "\". Le chemin d'accès par défaut, composé d'une unité de disque (lisible et modifiable par `chdrive`) et d'un chemin de répertoires (lisible par `dir$` et modifiable par `chdir`; il y a un chemin pour chaque disque), n'a pas besoin d'être précisé.

Exemple

Après exécution des commandes Basic :

```
chdrive "A"
chdir "\REP.EXX\BLEU.EYY"
```

le chemin d'accès par défaut est `A:\REP.EXX\BLEU.EYY`, `chdrive` vaut 1 (lecteur A), `dir$` vaut "\REP.EXX\BLEU.EYY", et le nom du fichier ci-dessus peut être simplifié en `COL.Z` ou même en `COL` dans les commandes comme `save` ou `merge` qui rajoutent l'extension Z. Les formes :

```
REBLEU\H.Y
\REBLEU\H.Y
..\VERT.EYY\GYT.Z
```

correspondent alors respectivement aux noms complets :

```
A:\REP.EXX\BLEU.EYY\REBLEU\H.Y
A:\REBLEU\H.Y
A:\REP.EXX\VERT.EYY\GYT.Z
```

En effet un "\" au début indique que le nom est donné à partir de la racine du disque et ".." fait revenir au répertoire parent.

Lorsqu'une commande demande le nom d'un fichier, Basic 1000d offre deux possibilités. On peut soit entrer le nom complet ou simplifié au clavier, soit cliquer sur la souris pour faire apparaître le sélecteur de fichier de l'AES. L'extension est Z par défaut pour les fichiers source, bibliothèque et help. Lors de l'écriture d'un fichier sur disque, une confirmation est demandée lorsqu'un fichier de même nom existe déjà. Ce fichier sera en effet recouvert par le nouveau.

∆ Si on veut sauvegarder la source sous forme ASCII où les lignes sont séparées par `chr$`(13) `chr$`(10), il faut utiliser une extension autre que Z. Un fichier Z est un fichier ASCII, mis à part que les lignes sont séparées par `chr$`(0). Les

fichiers lus dans la source peuvent être de type ASCII ou Z, sans que l'extension suive nécessairement les règles d'écriture ci-dessus. Basic 1000d se charge de la conversion au type Z, qui est la représentation interne utilisée. En plus de cette conversion, les espaces en début et fin de ligne sont supprimés. Les fichiers lus dans la bibliothèque ou help doivent être du type Z

Exemple: disque virtuel

Supposons que nous ayons installé un disque virtuel D dans le bureau. Pour pouvoir lire ou écrire des fichiers sur ce disque D, nous sélectionnons le disque D avec la commande Basic :

```
CHDRIVE "D"
```

ou

```
CHDRIVE 4
```

Après cela, nous fixons le répertoire courant avec la commande Basic :

```
CHDIR "REP.EXX"
```

où on notera que l'on écrit le chemin entre guillemets, à la différence de la commande DIR de l'éditeur qui est décrite plus bas. Les opérations de sauvegarde ou lecture auront alors lieu dans le disque virtuel. Pour utiliser de nouveau le lecteur A, exécuter :

```
CHDRIVE "A"
```

Commandes disques (menu FILES)

Les commandes décrites ici se trouvent dans le menu FILES, sauf Save et Merge, qui sont directement accessibles à partir des menus Basic et Edit Source. Seules les commandes éditeur sont étudiées ici, les commandes Basic comme `chdrive` pour changer de disque seront étudiées ultérieurement.

DIR (F1)

Liste les fichiers d'un répertoire. La commande demande le chemin `path` de la façon suivante :

```
ENTRER LE CHEMIN OU LIGNE VIDE ( EXEMPLE  A:\DOC\ ) >
```

Ecrire alors `path` sans guillemets.

Si `path` est vide, on prend le répertoire par défaut.

Si `path` se termine par "\" ou ":" le Basic rajoute `*.*` après `path`, ce qui demande la liste de tous les fichiers du répertoire.

Exemple

Après avoir sélectionné Dir et entré `b:*.z` pour `path`, on obtient :

```
[ A:\REP.EXX\BLEU.EYY ] b:*.z    458752 octets libres
```

```
3360 PUBLIC.Z
```

La commande a donné d'abord, entre crochets, le disque et chemin de répertoire par défaut, puis `path`. Elle a écrit ensuite la taille disponible sur le disque B. La taille disponible n'est écrite que dans le cas des disques A ou B, et il s'agit du disque dans `path` s'il est précisé.

Ensuite, la commande a listé tous les fichiers qui vérifient le nom `path` (y compris les fichiers cachés) avec leur taille. Ici il n'y en avait qu'un. Si on avait donné `b:` au lieu de `b:*.z`, `b:` aurait été transformé en `b:*.*` qui liste tous les fichiers du disque B.

KILL (F2)

Efface un fichier. Donner le nom avec l'extension.

MERGE (F11)

Cette commande des menus Basic ou Edit Source, rajoute un fichier à la fin de la source.

MERGE,M (F6)

Insère le fichier devant la ligne M de la source, qui doit être choisie avant l'appel de la commande.

LOAD (F10)

Vide la source et y charge le fichier.

MERGE,L (F8)

Cette commande, aussi dans le menu LBR/HLP, rajoute le fichier à la fin de la bibliothèque.

LOAD,L (F9)

Cette commande, aussi dans le menu LBR/HLP, vide la bibliothèque et y charge le fichier.

MERGE,H (F18)

Cette commande, aussi dans les menus LBR/HLP et HELPS, rajoute le fichier à la fin de help. Le nom du fichier peut être autre que HELP.Z.

LOAD,H (F19)

Cette commande, aussi dans les menus LBR/HLP et HELPS, vide help et y charge le fichier, qui doit s'appeler HELP.Z. Si help est vide, l'appui sur la touche Help effectue d'abord cette commande.

LOAD IMG (F3)

Lit un fichier et le charge dans la mémoire. L'extension est IMG par défaut. La commande demande les adresses DEBUT et FIN, puis charge le fichier à partir de l'adresse DEBUT, mais sans écrire en FIN ni au delà. Il y a ainsi chargement de `min`(FIN-DEBUT, L) octets où L désigne la longueur du fichier. Pour lire le fichier en entier, on peut donner -1 pour FIN.

SAVE (F12)

Cette commande des menus Basic ou Edit Source, écrit toute la source.

Δ L'écriture est beaucoup plus rapide lorsque la source commence à une adresse paire. C'est le cas lorsque la bibliothèque et Help sont vides.

SAVE,B (F7)

Cette commande, aussi dans le menu BLOCK, écrit le bloc, qui doit être défini avant l'appel de la commande.

SAVE IMG (F4)

Ecrit une zone mémoire sur disque. L'extension est IMG par défaut. La commande demande les adresses de début et de fin (c'est à dire le premier octet à ne pas écrire).

VERIFY (F5)

Cette commande relit le dernier fichier lu ou écrit, et vérifie que rien n'a changé. La vérification est impossible pour les fichiers source de type autre que Z.

Utilitaires (menu TOOLS)

POKE (F1)

Lit et modifie la mémoire. Il faut d'abord entrer au clavier :

```
[.s] [a]
```

.s

indique la taille et vaut `.B` (octet) `.W` (mot) ou `.L` (long). La valeur par défaut est `.B`. Ces valeurs peuvent s'écrire en appuyant sur Control et une des touches B, W ou L.

a

adresse de départ (par défaut, la dernière adresse utilisée par TOOLS). La commande indique la valeur qui va être recouverte par le poke, et attend l'entrée de la première valeur à poker (validée par CR), puis des valeurs suivantes. Sortir de la commande par Break, ou en cliquant un bouton, ou en entrant une expression illégale, par exemple une ligne vide.

QUERY (F2)

Examine la mémoire. La commande attend une entrée, sous la même forme que pour poke. Elle donne le contenu du bloc de 128 octets commençant à l'adresse `a`.

COPY (F3)

Copie d'un bloc mémoire. La commande demande les valeurs de DEBUT, FIN et DEST, puis recopie les octets de DEBUT à FIN – 1 en DEST.

FILL (F4)

Remplissage mémoire. La commande demande la taille (voir `.s` ci-dessus dans Poke), DEBUT, FIN et DATA, puis remplit le bloc mémoire de DEBUT à FIN − 1 avec la valeur DATA, par octets, mots ou mots longs suivant la taille.

COMP (F5)

Comparaison mémoire. La commande demande DEBUT, FIN et DEST, puis compare le bloc mémoire de DEBUT à FIN − 1 avec le bloc commençant en DEST. Elle écrit les valeurs et adresses des octets différents dans les deux blocs.

Par exemple, pour comparer le contenu de deux fichiers, on peut libérer d'abord la mémoire au dessus de $50000 par la commande Basic

```
limit $50000
```

puis charger le premier fichier en $50000, le deuxième en $\$50000 + L$ (où L est la longueur du premier fichier). La commande Comp donne ensuite les différences entre les deux fichiers.

QUIT (F9)

Sortie du Basic 1000d.

MEM MAP (F10)

Carte mémoire. Cette commande peut s'appeler dans la fenêtre Basic, en tapant :

```
= [CR]
```

On obtient par exemple :

```
Bspg  010AD8
Help  0391C6           0
Lbry  0391C8           0
Src   0391CA      498795
Limit 0F4000
Himem 0F4000
Proc        20000           100%
Xqt          4096           100%
Fre        232550            24%
```

La commande donne en hexadécimal les adresses de la page de base, `basepage`, des fichiers Help, Bibliothèque et Source, le haut du Basic, `limit`, et le haut de l'allocation, `himem`.

En décimal, elle donne aussi les longueurs des fichiers (ici Help et Bibliothèque sont vides et la Source prend 498795 octets).

La ligne Proc indique le nombre d'octets disponibles dans la pile des procédures et boucles, ainsi que sa valeur rapportée à `s_pro` (c'est 100% après `clear`). La ligne Xqt donne le nombre d'octets encore libres dans la pile des exécutions et sa valeur rapportée à `s_xqt`. La ligne Fre donne le nombre d'octets non utilisés, ainsi que sa valeur rapportée à l'allocation.

NEW (F11)

Efface la source et ouvre la fenêtre Edit Source.

OLD (F12)

La commande essaie de récupérer la source après NEW. Elle peut également permettre de retrouver une source éditable après une destruction.

Aides (menu HELPS)

Appel d'aides à la programmation

AUTHOR (F1)

Le numéro de version du Basic et l'auteur

```
    Basic 1000d
   version  1.00
 J.J. LABARTHE 1990
```

KEYBRD (F2)

Les codes que renvoient les fonctions `keyget` et `keytest`

EDITING (F3)

Rappel des touches particulières en édition

ASCII (F4)

Table des codes ASCII et des touches Alternate+lettre

PRINTING

Codes 0–$1F actifs dans `print`

NOHELP (F10)

Efface Help

MERGE,H (F18)

Rajoute un fichier à la fin de help (voir le menu FILES)

LOAD,H (F19)

Vide help et y charge le fichier HELP.Z (voir le menu FILES)

HELP (F20)

Cette commande, identique à l'appui sur la touche Help, ouvre la fenêtre Help (elle charge HELP.Z si la fenêtre est vide).

Imprimer la source (menu PRINTER)

Le menu PRINTER contient la commande Print (F16) qui imprime toute la source, et la commande Print,b (F17) qui imprime le bloc.

L'instruction `page` insérée dans la source permet de mettre un titre en haut de chaque page du listing. La sortie imprimante de la source se fait avec les conversions suivantes :

code dans la source	envoi
0	13, 10
$n \in [1, 127]$	n
$n > 127$	$\texttt{peekb}(\texttt{systab} + 90 + (n - 128))$

La table de 128 octets en $\texttt{systab} + 90$ est la table de conversion des caractères $\geq$ 128. Cette table convient aux imprimantes Epson, les symboles Alternate+Lettre étant rendus par la lettre majuscule en italique.

En outre, pour un fonctionnement correct, il faut définir les variables d'état `page_width` et `page_length` avec les valeurs de l'imprimante.

Exemple

Pour imprimer sur 132 colonnes et 50 lignes, exécuter d'abord :

```
page_width 132
page_length 50
```

Chercher et changer (menu FND/CHG)

Modifier si nécessaire les paramètres (F1 à F7) puis effectuer la recherche ou le changement (F8 à F10). Noter que les touches de fonctions avec et sans Shift sont équivalentes dans ce menu.

DEFINE S (F1)

Permet de définir la chaîne S. Elle est formée d'au plus 48 caractères. Les caractères de codes 1 à 10 ne peuvent être recherchés. Les caractères suivants

ont un sens particulier :

ƒ	[a] S	séparateur
æ	[a] Z	zéro
í	[a] F	filler (chaîne de remplissage)

Le symbole ƒ ([a] S) désigne un caractère autre que les lettres non accentuées, les chiffres et les symboles !#$%.?_âçéèêîô. Le symbole æ ([a] Z) désigne la séparation entre 2 lignes de la source (octet nul). Le symbole í ([a] F) désigne n'importe quelle chaîne, éventuellement vide, incluse dans une ligne source donnée. [a] F doit être précédé et suivi d'un caractère différent de [a] F. Si la chaîne S se termine par des espaces, entrer la chaîne entre deux guillemets.

Exemple

Si la source contient les deux lignes :

```
W=MO+X
PRINT MOD(X^60-1, X^41-1, X)
```

la chaîne ƒ`MO`ƒ (`MO` précédé et suivi de séparateurs) se trouve seulement dans la première ligne. La chaîne ƒ`WíæíMOD` se trouve aussi dans cet exemple.

DEFINE T (F2)

Permet de définir la chaîne T, avec les mêmes règles que pour la chaîne S.

EXG (F3)

Permet d'échanger les chaînes S et T. Seulement le contenu des chaînes S et T est redéfini, aucune modification n'intervenant dans la source. Il est parfois possible d'annuler un changement intempestif de S en T simplement en répétant l'ordre de changement, après avoir permuté les chaînes S et T.

WHERE (F4)

La recherche ou changement peut être effectuée dans toute la source (ALL) ou seulement dans le bloc (A–B).

REPEAT (F5)

La recherche ou changement continue (Y) ou non (N) après la première occurrence.

DISTINGO (F6)

La recherche ou changement fait la distinction (Y) ou non (N) entre les majuscules et minuscules.

HOLD (F7)

La recherche ou changement s'arrête (Y) ou non (N) après impression d'une page d'occurrences (16 lignes).

FIND S (F8)

Recherche la chaîne S

FIND T (F9)

Recherche la chaîne T

CHANGE S->T (F10)

Change la chaîne S en T

La chaîne S (resp T) définit les zones 1, 2, ... (resp 1', 2', ...) :

[a] S 1111111 [a] S ... [a] F 222222 [a] F 333333 [a] S ...

Ces zones ne contiennent ni [a] S ni [a] F mais peuvent contenir [a] Z. Dans chaque région séparée par les [a] F il y a une seule zone qui commence au premier caractère ou au deuxième si le premier est [a] S. La commande de changement exige que le nombre de zones de S et T soit le même. Elle effectue alors le remplacement de la zone 1 par la zone 1', de la zone 2 par la zone 2' ...

Exemple

Taper sur F1 et entrer ":" comme chaîne S. Taper sur F2 et entrer ":æ" (le deuxième caractère est [a] Z) comme chaîne T. Taper éventuellement sur F4, pour que le changement ait lieu dans le bloc. En tapant sur F10 on lance le changement de S en T, ce qui coupe toutes les lignes du bloc contenant ":" en deux après ":".

Exercice Apluslong

Comment peut on remplacer le label `A` par le label `apluslong`? La difficulté vient de ce que beaucoup de `A` ne doivent pas être changés. (L'utilisation d'un tel label est très malencontreuse).

7

Exécution des programmes



Un programme est formé de lignes de forme générale :

`[label:] [instruction] [' commentaire]`

Il y a au plus une instruction par ligne. La référence à une ligne dans le programme (par exemple dans `goto`) doit obligatoirement se faire à l'aide d'un label. L'apostrophe non entre guillemets indique le début d'un commentaire.

Programmes de contrôle

Ce sont des procédures que vous pouvez écrire en Basic, et qui contrôlent l'exécution des autres programmes. Les noms de ces procédures sont prédéterminés et doivent être écrits en majuscules. Ces sous-programmes peuvent se trouver dans la source ou la bibliothèque, ou être absents.

B_INIT

B_END

Basic 1000d exécute la procédure B_INIT au début de RUN (F8) et DEBUG (F9), et la procédure B_END lors du retour à la fenêtre Basic (après `stop`, `end`, Break ou erreur).

Exemple

La procédure B_INIT ouvre complètement la fenêtre RUN, et le nombre calculé par le programme est affiché sur la première ligne de l'écran. L'instruction `message` du programme est exécutée dans cette fenêtre, mais celle de la procédure B_END est effectuée après le rétablissement du menu Basic. Lorsque une instruction est effectuée en mode direct, les procédures B_INIT et B_END sont ignorées.

```
       print 10^10
       message "Fin du programme"
       stop
B_INIT:cursh 0
       cls
       return
 B_END:message "Retour au Basic"
       return
```

B_TRACE

La procédure B_TRACE est exécutée avant chaque instruction (sauf dans les programmes B_INIT, B_END, B_DEBUG ou B_TRACE) pendant RUN (F8) ou DEBUG (F9)

Exemple

En utilisant le numéro de ligne `nextline` et l'adresse du code `nextcode`, la procédure B_TRACE écrit le numéro et le texte de chaque ligne exécutée. La commande `breakpoint` enclenche le débogage lorsque `I` prend la valeur 14. Noter qu'il est nécessaire de définir le type de `I` (on aurait pu aussi le définir dans une procédure B_INIT), sinon il serait pris de type littéral.

```
        for I=1,20
          print I
        next I
        stop
B_TRACE:index I
        print nextline;">";peekz$(nextcode)
        ift I=14 breakpoint
        return
```

Sortie

```
1>for I=1,20
2>print I
1
...
```

B_DEBUG

La procédure B_DEBUG est exécutée à la fin de la sortie du listing dans la fenêtre DEBUG (F9, F19 ou F29), c'est à dire avant l'attente de la commande de débogage. Si elle contient des instructions `print`, l'écriture se fait dans la fenêtre DEBUG. Elle peut être utilisée pour suivre des variables, sans interférer avec l'écran de sortie.

Exemple

Lorsque ce programme est lancé par RUN (ou DEBUG), la procédure B_INIT est d'abord exécutée, ce qui impose le débogage par suite de la commande `breakpoint`. Ensuite, B_TRACE est appelé, la fenêtre DEBUG est écrite, B_DEBUG est appelé et le débogueur attend une commande avant d'exécuter la première ligne. Si le pas à pas est utilisé (appui sur Return), le programme B_DEBUG écrit dans la fenêtre DEBUG le texte de la ligne qui vient d'être exécutée et la valeur de la variable `v`. Noter qu'il est nécessaire de déclarer les variables (dans B_INIT ou B_TRACE) et que lors de l'exécution de B_DEBUG la variable `d` contient déjà le texte de la ligne suivante.

```
        forv v in (5,[11,13,1])
          print v
        nextv
        stop
B_TRACE:c=peekz$(nextcode)
        exg c,d
        return
B_DEBUG:print "La ligne précédente était ";c
```

```
        print "v=";v
        return
B_INIT:char c,d
        var v
        breakpoint
        return
```

B_USER

La procédure B_USER est appelée en cliquant sur la case B_USER (F7) de la fenêtre Basic ou Edit Source. Cet appel est équivalent à l'appel en mode direct par `B_USER` et n'est donc pas précédé de `clear`.

Exécution (Run)

RUN (F8)

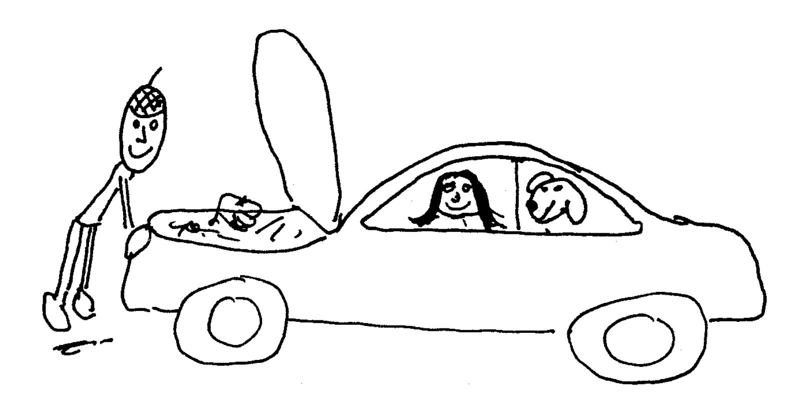
Lance l'exécution du programme dans la source, ce qui produit les effets suivants. Une instruction `clear` efface tous les anciens labels et variables. Ensuite Basic 1000d effectue un premier passage pour construire la table des labels. Il parcourt pour cela la bibliothèque puis la source, chacune jusqu'à une instruction `end` où jusqu'à la fin s'il n'y en a pas. Le programme B_INIT est appelé s'il existe. Les instructions sont exécutées en partant de la première ligne de la source (le programme B_TRACE s'il existe est appelé avant chaque instruction). L'exécution s'arrête sur une commande `stop`, `end` ou une erreur, ou encore ad libitum par Break. Il y a alors retour dans la fenêtre Basic, puis appel du programme B_END s'il existe.

RUN... (F18)

Permet de faire partir l'exécution d'une ligne quelconque. Il faut entrer la ligne (même syntaxe que dans la commande `#ligne`). A la différence de RUN (F8), un `clear` n'est pas effectué, sauf si la source a été trop modifiée (en détruisant par là les variables du programme). De plus, les programmes B_INIT, B_TRACE et B_END sont ignorés dans ce mode d'exécution.

8

Mise au point des programmes



Débogage

Le débogueur permet de contrôler l'exécution d'un programme. L'écran du programme n'est pas modifié de sorte que l'exécution par le débogueur produit les mêmes résultats que l'exécution par RUN.

Entrée dans le débogueur

DEBUG (F9)

Correspond à RUN (F8).

Les programmes B_INIT, B_TRACE et B_END sont exécutés sans contrôle du débogueur.

DEBUG... (F19)

Correspond à RUN... (F18).

Permet d'exécuter une portion de code, à partir d'une ligne quelconque et sans réinitialisation.

DEBUG+ (F29)

Permet de reprendre l'exécution d'un programme, après retour à la fenêtre Basic. L'écran du programme est alors rétabli. On peut à l'aide de cette commande poursuivre l'exécution d'un programme arrêté par Break ou par une erreur, après correction de l'erreur.

Ecran de débogage

Il contient le damier DEBUG, et une dizaine de lignes source. Ce sont les lignes à partir de la ligne à exécuter. Remarquer que ces lignes sont numérotées à partir de 1, en vue de faciliter le débogage. Le véritable numéro n de la première ligne écrite est indiqué par :

```
le prgr va exécuter la ligne n
```

En tête de ces lignes peut se trouver une ligne (xqt), qui écrit en clair l'instruction qui va être exécutée (en cas de `@`, ou pour l'instruction `xqt`, ou pour la commande du `ift` si la condition est vraie). Rappelons que l'écran peut être modifié par le programme de contrôle B_DEBUG.

Actions possibles

Cliquer le bouton gauche sur le damier (ou touches F), ou taper une touche pour les commandes de débogage.

Cliquer le bouton droit sur une ligne du programme pour exécuter jusqu'à cette ligne (fait une boucle si c'est la première ligne).

Cliquer le bouton gauche sur un nom de la page pour obtenir le type du nom et sa valeur si c'est une variable, index ou fonction. Si par exemple on clique sur `divr` dans `divr(psi(7),@3)` l'expression est calculée (où `psi` est une

fonction écrite en Basic). Noter toutefois que le cliquage d'expressions contenant `pop` ou `pop$` va ôter un élément de la pile et perturber le programme, et que cliquer sur une fonction ayant en argument des noms pas encore définis produit une sortie erreur.

Commandes de débogage

SSal (F1) ou touche Return

Exécute l'instruction de la ligne 1 et s'arrête sur l'instruction suivante. C'est le pas à pas. En appuyant seulement sur Return (ou F1) on peut ainsi suivre le programme dans toutes ses instructions.

SSsr (F11) ou barre d'espace

Exécute le pas à pas seulement dans dans la source. Les procédures de la bibliothèque sont alors invisibles (elles sont visibles dans le pas à pas F1)

Arret (F2) ou touche A

Retour à la fenêtre Basic. Il est alors possible d'effectuer des commandes de l'éditeur, des instructions Basic en mode direct et des modifications de source, puis de revenir dans le débogueur par DEBUG+ (F29) et de continuer le programme. Toutefois les modifications de source ne doivent pas être trop importantes (destruction des tableaux). Elles ne doivent pas non plus modifier l'adresse de l'instruction de retour si on était dans une procédure ou une boucle. La ligne qui était à exécuter devient la ligne M, de sorte que l'on peut examiner facilement le programme en ouvrant la fenêtre Edit Source (F10).

Break (F3) ou touche B

Le débogueur demande un BREAKPT (point d'arrêt). Il faut indiquer au programme une ligne en tapant au clavier `ligne` puis Return. On peut donner cette ligne en utilisant les labels (syntaxe de la commande `#ligne`). Le programme s'exécute ensuite jusqu'à la ligne spécifiée.

Brks (F4) ou touche C

Le débogueur demande des points d'arrêts (BREAKPT). Il faut entrer au clavier plusieurs fois `ligne`. Après le dernier point d'arrêt taper Return pour valider la commande. On peut utiliser jusqu'à 16 points d'arrêt. Le programme s'exécute jusqu'au premier point d'arrêt atteint.

RTS (F5) ou touche D

Exécute jusqu'à sortir du sous-programme en cours. Par exemple :

```
S:GOSUB B
A:W=DER(W,X)
  stop
B:W=SUM(I=1, 10 OF Y^I)
  W=SUBS(W, Y=X)
  return
```

Après DEBUG le débogueur va exécuter la ligne S. Appuyez sur Return, le débogueur va ligne B. Appuyez sur D, le débogueur va ligne A.

SCRN (F6) ou touche E

Remet l'écran du programme, ses couleurs, sa résolution et son curseur. Vous pouvez alors admirer à loisir ce que sort votre programme. Attend ensuite une commande de débogage. Seules les commandes du damier indiquées sont valables. Pour les autres commandes il faut d'abord revenir à l'écran de débogage par NOP (F10 ou touche K). Noter que la touche Clr (ou F21) permet d'effacer l'écran du programme.

FOIS (F7) ou touche F

Demande le nombre de FOIS et un point d'arrêt BREAKPT (on peut rentrer Return sans rien d'autre pour la ligne 1). Le débogueur lance le programme et s'arrête lorsqu'il a trouvé la ligne BREAKPT le nombre de fois donné. L'instruction de la ligne BREAKPT aura donc été exécutée FOIS − 1 fois (ou FOIS fois si BREAKPT est la ligne 1)

CYCL (F8) ou touche H

Exécute jusqu'au retour à l'instruction de la ligne 1. Cette commande équivaut donc à la commande F avec FOIS=1 et BREAKPT=(vide) [Return]. Cette commande peut aussi être effectuée en cliquant sur la ligne 1 avec le bouton droit.

RUN (F9) ou touche I ou J

Continue le programme sans s'occuper de l'arrêter. On revient à la fenêtre Basic lorsque le programme trouve l'instruction `stop`, `end` ou sur erreur, ou après Break. On revient dans le débogueur sur l'instruction `breakpoint` insérée dans la source.

NOP (F10) ou touche K

Récrit seulement la page de débogage. Utile lorsqu'on a cliqué avec le bouton gauche sur un grand nombre de noms, ou pour revenir de l'écran du programme après SCRN (F6 ou touche E).

Lg2 (F12) à Lg9 (F19) ou touches 2 à 9

Lance le programme et retourne au débogueur avant l'exécution de la ligne 2 (ou 3, ..., 9) du listing du débogueur. Cela revient donc à effectuer la commande Break (F3) pour une des lignes listées.

Cette commande peut aussi s'effectuer en cliquant le bouton droit sur la ligne voulue.

LOOP XIT (F20) ou touche L

Si la ligne à exécuter se trouve dans une boucle `for` ... `next`, `forv` ... `nextv`, `forc` ... `nextc`, `do` ... `loop` ou `repeat` ... `until`, lance le programme et s'arrête après la sortie de la boucle.

Dans le cas d'une boucle `while` ... `wend`, la commande fait revenir sur `while`. Pour arrêt après sortie de la boucle, mettre un breakpoint après le `wend`.

ARG (F22) ou touche `@`

Si on se trouve dans un sous-programme on obtient le k-ième argument en entrant k, et le nombre d'arguments en entrant 0.

Exemple de débogage

```
I10:deb=5
I11:cof=cof+x^deb
    deb=deb-1
    ift deb>0 goto I10
I12:print coff
    for i=1,100
      cof=cof+y(i)^2+y(2*i)
    next i
    print cof
    stop
  y:function(i)
    value=(z-i)/i
    i=i-1
    ift modr(i,5)=0 value=1
    return
```

Ce programme boucle par erreur. Si vous lancez le programme par RUN, vous ne comprenez pas pourquoi rien ne sort.

Indiquons comment mettre au point ce programme `I10` par le débogueur. On part de la première ligne `I10` par DEBUG (F9). Le débogueur se positionne sur la première instruction. Il est ensuite conseillé d'exécuter chaque instruction au moins une fois en mode pas à pas. Appuyez sur Return, la première instruction a été exécutée, le débogueur est positionné sur la deuxième ligne. Appuyez encore 3 fois sur Return (ou sur F1). Vous pouvez aussi cliquer, au cours de ce pas à pas, sur les divers noms écrits sur la page avec le bouton gauche. Le débogueur revient ligne `I10`, alors que vous auriez voulu aller ligne `I11`. Voilà l'erreur.

Retournez à la fenêtre Basic par Arret (F2 ou touche A), puis ouvrez la fenêtre Edit Source, changez `goto I10` en `goto I11` et relancez le débogage par DEBUG (F9), mais maintenant vous voulez aller jusqu'en `I12` rapidement. Appuyez sur 5 seulement (la ligne `I12` est numérotée par 5) ou cliquez le bouton droit sur la ligne `I12.` Le débogueur exécute la boucle (presque) en vitesse réelle et se positionne en `I12`.

Vous observez maintenant que vous vouliez écrire `cof` et non `coff`. Retournez à la fenêtre Edit Source et corrigez `coff` en `cof`. Vous pouvez maintenant reprendre le programme au point où il était, sans réexécuter depuis le début, à l'aide DEBUG+.

Taper Return, le `print` est exécuté. Pour voir l'écran du programme appuyez sur E ou F6. Pour revenir au débogueur appuyez sur K ou F10.

Continuez le pas à pas. Observez que le calcul de l'expression

```
y(i)^2+y(2*i)
```

vous fait parcourir deux fois la fonction `y`. Au fur et à mesure que vous corrigez les erreurs, vous n'avez plus besoin d'exécuter toutes les instructions une à une. Le débogueur offre de nombreuses possibilités pour sauter les parties de programme déjà au point. Ainsi, vous pouvez revenir de la fonction `y` en tapant sur D ou F5, et sortir de la boucle `for` ... `next` en tapant sur L ou F20.

Erreurs de syntaxe

Les erreurs de syntaxe sont clairement indiquées. L'écran du programme avant l'erreur est sauvegardé, ce qui permet, après correction de l'erreur, de continuer l'exécution du programme (par la commande éditeur Debug+). La ligne en erreur est sortie avec son numéro de sorte que la correction de la source est facilitée. L'endroit où l'erreur a été détectée est marqué sur une autre ligne avec un point d'interrogation. La cause de l'erreur est en général juste avant ou après ce point d'interrogation. Remarquez que l'erreur dans une instruction comme `read` peut être causée par une erreur dans une autre ligne. Dans ce cas la ligne avec le point d'interrogation diffère de la ligne numérotée, et il faudra rechercher dans la source la ligne en erreur.

Si vous ne comprenez pas d'où vient l'erreur de syntaxe, vérifiez le type du nom qui a produit l'erreur ainsi que son contenu.

Exemple

Supposons qu'un programme produise l'erreur :

```
*ERREUR* INDEX/ADRESSE
t_type t ?
t_type t
```

La commande, qui définit le type de texte graphique, vous semble correcte, vous examinez le type et le contenu du nom `t` en exécutant en mode direct :

```
type t
print t
```

Les réponses respectives sont :

```
t est de type lit
  t
```

Ainsi, vous avez peut-être oublié de donner à `t` sa valeur :

```
t=3
```

par exemple, et `t`, qui était indéfini a été considéré comme étant un symbole. Mais l'erreur était peut-être que vous vouliez écrire :

```
t_type T
```

où `T` avait bien une valeur entière.

Les erreurs sont bien sûr encore plus difficiles à découvrir lorsqu'elles ne provoquent pas d'erreurs de syntaxe. En pratique, il est indispensable d'utiliser le débogueur pour exécuter tout nouveau programme et en éradiquer les erreurs de logique.

Lorsqu'une erreur se produit dans un programme de la bibliothèque, la ligne en erreur n'est pas numérotée. Dans ce cas, on peut exécuter de nouveau le programme, avec la bibliothèque chargée dans la source, pour localiser l'erreur.

Messages d'erreur

Sautez cette section en première lecture, revenez-y quand vous voudrez des indications supplémentaires sur les messages d'erreurs. Nous donnons la liste complète des messages d'erreur et des indications sur les corrections à apporter au programme dans les cas les plus difficiles.

`@ DANS MAIN`

`@k` ne peut être utilisé que dans un sous-programme.

`ADRESSE IMPAIRE`

L'argument attendu doit être un entier*32 pair. Par exemple, l'adresse de `pokew` est impaire dans :

```
pokew 1,0
```

`AES/VDI`

Erreur d'arguments, mauvais numéro de fonction, trop d'arguments, etc. lors des commandes `vdi` et `aes`. Par exemple, il y a trop d'arguments dans :

```
vdi 32,1,1
```

`APPEL AES AVEC UNE RESOLUTION AUTRE QUE CELLE DU BUREAU`

Pour éviter de nombreuses anomalies, comme par exemple une souris confinée dans la moitié de l'écran, les appels AES doivent être effectués en résolution `resolution0`.

`APPELER PRFACT EN MODE DEVELOP`

On ne peut pas utiliser le mode `factor` avec prfact.

`ARGUMENTS DE FONCTION`

Mauvais nombre d'arguments d'une fonction du Basic, par exemple :

```
print xbios(4,1)
```

`BLOC`

Erreur éditeur. Bloc non défini, ou essai de déplacer le bloc devant une de ses lignes internes.

`CANAL DEJA OUVERT`

Il faut d'abord fermer un canal ouvert avant de pouvoir le rouvrir. Par exemple, essai d'ouvrir deux fois le canal 1 par `open` :

```
open "i",1,"nul:"
open "i",1,"x.x"
```

`CANAL NON DE TYPE R`

Essai d'utilisation d'un canal comme fichier à accès sélectif sans ouverture par `open "R"`, par exemple :

```
open "i",1,"nul:"
field #1,8 as c$
```

`CANAL NON OUVERT`

Avant d'être utilisable, un canal doit être ouvert. Par exemple, s'il manque `open "o",1,...` :

```
print #1,x
```

`CANAL NON OUVERT EN ECRITURE`

Essai d'écriture sur un canal d'entrée. Par exemple, essai d'écriture sur un canal d'entrée (celui d'`input`) :

```
print #101,1
```

`CANAL NON OUVERT EN LECTURE OU "NUL:"`

Essai de lecture sur un canal de sortie ou de type "NUL:". Par exemple, essai de lecture sur un canal d'écriture (celui du `print`) :

```
input #102,x
```

`CANAL SANS POINTEUR`

Essai d'utilisation d'un pointeur dans un canal de type "LST:", "VID:", etc. Par exemple pour le canal 101 ("VBS:"), où on demande la position du pointeur :

```
print lof(101)
```

`CHAINE PRISE POUR VARIABLE`

La vérification qu'une zone mémoire représente une expression mathématique échoue. Dans l'exemple

```
push$ "a"
w=pop
```

`pop` ne trouve pas une expression acceptable sur la pile.

`CHECK SUM`

Le rappel de l'écran par `rscreen` doit utiliser une chaîne créée par `screen$`. Par exemple, la vérification échoue dans l'appel avec une chaîne vide :

```
char c$
rscreen c$
```

`COMPARAISON`

Par exemple, si `A` n'est pas un réel :

```
ift A>2 stop
```

`COND`

La forme de la condition est illégale, par exemple le polynôme $1+x^2$ ne contient pas le littéral y dans :

```
cond 1+x^2,y
```

`DEBUG`

Erreur lors de l'appel du débogueur à partir de l'éditeur. On ne peut plus reprendre l'exécution (programme trop modifié), ou on indique une ligne de départ illégale.

`DEVELOPPEMENT EN X^-K`

Dans `str$`(w,/x), si w n'est pas une somme de la forme $\sum_i a_i x^i$, où i est entier relatif et a_i est indépendant de x, comme par exemple dans :

```
print str$(1/(1+x),/x)
```

`DIM/SIZE MODIFIE`

Par exemple :

```
index t(10)
index t(11)
```

Par contre il est admis de redéfinir un tableau avec les mêmes dimensions.

`DISQUETTE`

Problème lors d'une lecture/écriture.

`DIVISION`

Exemple qui conduit à une division par 0 :

```
w=mdpwre(2,-1,4)
```

`DOMAINE DE DEFINITION`

Un argument a une valeur en dehors des valeurs acceptées par la commande ou fonction. Par exemple le numéro de canal est trop grand dans :

```
open "i",250,"x.x"
```

`ENSEMBLE`

Ensemble mal écrit, par exemple dans :

```
ift 1 in (a
```

`ENSEMBLE ILLEGAL POUR BOUCLE`

Par exemple l'ensemble est infini dans :

```
forv v in [1,5]
nextv
```

`ENTIER*S`

La taille de l'entier est incorrecte. Par exemple (il faut un entier dans $[-128, 127]$ pour `pokebs`) :

```
pokebs $200,128
```

`EOL DOIT ETRE 0 OU 13,10`

Les fichiers lus par `(line) input #n` doivent coder la fin de ligne soit par l'octet 0 (fichier Z), soit par les deux octets 13, 10 (fichier ASCII). Le codage par `chr$`(13) seul n'est pas admis.

`EXCEPTION 68000`

Une erreur détectée par le micro-processeur.

`EXPONENTIELLE`

Exposant plus grand ou égal à 2^{15}. Par exemple :

```
w=2^(2^15)
```

`EXPOSANT TROP GRAND`

Lors d'un calcul, l'exposant d'un polynôme dépasse 2^{16} comme dans :

```
w=x^(2^14)
```

```
w=w^4
```

`EXPRESSION ILLEGALE`

Echec dans le décodage d'une expression, par exemple :

```
print 2^
```

`EXPRESSION LOGIQUE`

Les expressions logiques doivent être des entier*32. Par exemple :

```
w=2^33 and 1
```

`FATALE`

Une anomalie dans les données (zone dynamique des variables) a été détectée. L'accès aux données est devenu impossible et le Basic effectue un `clear`. Cette erreur peut survenir après une erreur mémoire, après un poke hasardeux.

`FICHIER HELP SANS \A`

Un fichier Help doit comporter des commandes de mise en page \A, \B , etc. et en particulier au moins une commande \A.

`FICHIER PAS TROUVE`

Lors d'une demande de lecture sur disque.

`FIN DE BOUCLE SANS DEBUT`

Par exemple `wend` non précédé de `while`.

`FIN DE FICHIER`

Essai de lecture au delà de la fin de fichier.

Exemple

Le programme suivant est correct et lit la totalité du canal 1.

```
c$="abc"
open "i",1,"mem:",c$
do
  ift eof(#1) exit
  print inp(#1);
loop
```

Sortie (50 ms)

```
 97  98  99
```

Par contre, si on omet la ligne `ift eof(#1) exit` qui arrête la lecture du fichier, on provoque cette erreur.

`FIN DE SOURCE`

Le programme ne trouve pas la fin d'une structure (`if` sans `endif` par exemple) et essaie de dépasser la fin de la source (ou la commande `end`). Ce type d'erreur est détectable en recherchant les défauts d'indentation de la source.

`FLOTTANT INTERDIT`

L'argument doit être exact, par exemple dans :

```
w=num(exp(1~))
```

`FONCT/PROC QUI TERMINE DES BOUCLES`

Cette erreur est détectée lors du retour de sous-programme, par exemple dans :

```
          for i=1,10
            p
        p:next i
          return
```

L'exemple ci-dessus n'est pas si simple qu'il paraît et il est instructif de l'exécuter avec le débogueur pour comprendre ce que fait Basic 1000d.

FOR/NEXT

Mauvaise structure détectée, `next` sans `for`, etc. Par exemple, le pas est nul dans :

```
for i=1 to 3 step 0
```

FORME SOURIS

Dans l'appel de `defmouse`, la chaîne doit avoir une longueur de 74 octets, ce qui n'est pas le cas dans :

```
char c$
defmouse c$
```

GEMDOS #-nn

Erreur système

HORS DU TABLEAU

La vérification des limites d'un tableau échoue. Dans l'exemple suivant, la taille du tableau `g` ne peut déborder celle de `f`.

```
var f(10)
local dataa f(0) access g(15)
```

IMBRICATION DES BOUCLES

Comme par exemple dans :

```
forv v in (1)
  forc c in ("a")
  nextv
nextc
```

INCOMPATIBLE AVEC EXIT

La commande `exit` ne doit pas provoquer le survol d'une instruction `next` regroupant plusieurs boucles, comme dans :

```
for k=1,1
  exit
  for i=1,2
    for j=1,2
  next j,i
next k
```

INCREMENT DE BOUCLE

L'incrément d'un ensemble discret ne peut être nul, comme par exemple dans :

```
forv v in [1/2,3/2,0]
nextv
```

INDEX/ADRESSE

L'argument attendu doit être un entier*16 ou *32. Par exemple (il faut un entier*16) :

```
c$=mki$($9876)
```

`INSERTION DE $19 OU $1A`

Erreur éditeur. Ces caractères qui marquent le début et la fin de source ne peuvent être édités. L'erreur peut se produire après chargement d'un fichier non de types ASCII ou Z dans la source.

`INSERTION INTERDITE`

Erreur éditeur. Par exemple après :

```
125.124
```

en mode direct (voir la commande `n.texte` de l'éditeur).

`INSTRUCTION ILLEGALE`

Nom de la commande mal écrit ou procédure absente. Ce message est aussi généré lorsque le Basic attendait un déterminant (mot clef interne à une commande comme `to` et `step` dans `for`) ou un séparateur (signe "=", etc.). La cause de l'erreur peut alors être bien avant l'endroit indiqué par "?". Par exemple, il manque le signe "=" devant `y` dans :

```
print subsr(1+x,x^2 y)
```

Une autre cause de cette erreur est la présence de données superflues à la fin de la commande, comme par exemple dans :

```
print 1)
```

`INTEGRATION`

Le dénominateur de la fraction à intégrer doit être factorisable en facteurs de degré un, pour la fonction d'intégration interne du Basic. Cette restriction est levée pour l'intégration par les fonctions de la bibliothèque Math. Exemple :

```
print intg((x^2+2)^-1)
```

`LABEL ABSENT`

Erreur éditeur, par exemple dans :

```
#xyz
```

s'il n'y a pas de label `xyz`.

`LABEL ILLEGAL`

Par exemple :

```
cont:procedure(p)
```

car `cont` est un mot clef du Basic. Si des labels sont en erreur, l'exécution ne peut même pas commencer. Par contre, certaines commandes comme `print` seront acceptées en mode direct.

`LABEL REPETE`

On ne peut pas répéter un label, par exemple :

```
phase:procedure
    ...
phase:function(index x)
```

`LIGNE`

Erreur éditeur. Par exemple si `kl` est un label :

```
run 1+kl
```

au lieu de :

```
run kl+1
```

`LIGNE DE DEPART`

Donnée de ligne incorrecte dans le débogueur.

`LIGNE TROP LONGUE`

Une ligne source ne peut dépasser 3600 caractères environ.

`LIMIT/HIMEM`

L'adresse fournie ne convient pas. Une autre cause est que `limit` doit être dans le programme principal, pas dans une fonction comme dans :

```
     ift init
     stop
init:function
     limit max
     return
```

`LIT/VAR DIMENSION NON DEFINIE`

Se produit avec un tableau ou fonction non déclaré, par exemple :

```
s(3)=1
```

`LITTERAL COMPLEXE NON DEFINI`

Dans une fonction nécessitant les nombres complexes, comme :

```
print log(-1)
```

l'erreur se produit si le littéral complexe n'est pas défini. Pour corriger, rajouter par exemple :

```
complex i
```

`LOCAL`

Il n'est pas possible de déclarer comme local au même niveau de sous-programme, deux fois le même nom.

```
local index a
local index a
```

`LONGUEUR D'ENREGISTREMENT`

La longueur d'enregistrement d'un fichier à accès sélectif, spécifiée par la commande `open`, n'est pas égale à la longueur totale des champs, définis par les commandes `field`. L'erreur est détectée soit dans la commande `field` si les champs sont trop longs, soit lors d'une lecture/écriture. Dans l'exemple suivant la longueur des champs (=4) est inférieure à la longueur d'enregistrement (=5).

```
open "R",1,"MEM:",C$,5
field #1,4 AS x$
put #1
```

`LONGUEUR DE CHAMP`

Lors de l'écriture d'un fichier à accès sélectif, le champ en cause (de type char) a une longueur différente de celle spécifiée dans `field`. Dans l'exemple suivant, au moment de l'écriture le champ `x$` a une longueur de 3 octets au lieu des 4 octets attendus.

```
open "R",1,"MEM:",C$,4
```

```
field #1,4 AS x$
x$="abc"
put #1
```

`MEMOIRE`

Cette erreur se produit lorsqu'une instruction n'a pas disposé d'assez de place mémoire. Essayer d'abord d'augmenter la valeur de `pack`. Par exemple, si `pack` vaut 10000 et si une instruction a besoin de 10002 octets de mémoire il est possible d'obtenir cette erreur, alors que `print fre`, exécuté après l'erreur, indique 500000 octets de libre. L'erreur s'est produite parce que le Basic constatant qu'il restait plus de 10000 octets utilisables, n'a pas effectué un nettoyage de la mémoire.

Il se peut qu'il n'y ait vraiment pas assez de place, même après un pack. Si une opération sur une chaîne trop longue est en cause, penser à utiliser :

```
cadd long,"..."
```

au lieu de :

```
long=long&"..."
```

qui nécessite deux fois plus de mémoire.

De même pour des variables contenant des expr gigantesques, les opérations `vadd`, `vmul`, etc. peuvent également faire gagner ce facteur 2.

Enfin, Nohelp (libère la place utilisée par Help), et ne garder que l'indispensable dans la bibliothèque.

`MEM_FILES TROP PETIT`

Il y a trop de fichiers virtuels "MEM:". Il suffit d'augmenter la valeur de la variable d'état `mem_files`.

`MENU`

Par exemple si cela produirait un menu plus large que l'écran.

`MONOME/LITTERAL`

L'argument attendu était un monôme ou un littéral, par exemple :

```
w=subs(x^2+1,x^2=1)
w=subsr(x^2+1,3*x^2=1)
```

où il faut un littéral (ligne 1) ou un monôme normalisé (ligne 2).

`N.`

Erreur éditeur, par exemple :

```
100000.print
```

s'il y a moins de 100000 lignes.

`NOM REPETE`

Chaque nom ne peut avoir qu'un seul type. Par exemple :

```
    char phase
phase:function
```

On ne peut répéter un littéral dans la substitution flottante :

```
w=fsubs(x^2+1,x=1~,x=2~)
```

`NOM RESERVE`

Par exemple, `print`, qui ne peut être utilisé comme littéral :

```
w=print
```

`NOM TROP LONG`

Au plus 32 caractères.

`NOMBRE COMPLEXE`

Dans une opération binaire (+ − ∗ / ^ \ `div` `mod`), si une expression est flottante, l'autre ne peut pas contenir des littéraux, mis à part le littéral complexe. Avec un exposant exact non entier, comme l'exposant est transformé en flottant, on peut aussi obtenir cette erreur, par exemple (`x` littéral) :

```
w=x^(1/2)
```

`NOMBRE D'INDICES`

Par exemple la variable `c` doit être utilisée avec 2 indices :

```
char c(4,3)
print c(1)
```

`NOMBRE NON PREMIER`

L'argument doit être un nombre premier, par exemple dans :

```
w=mdff(x^2+1,6)
```

`NOMBRE REEL`

Dans (`a` littéral) :

```
print abs(a)
```

l'argument doit être réel.

`nomi DE TYPE CHAR`

Un nomi de type char était attendu, par exemple :

```
index c
cadd c,"a"
```

`nomi DE TYPE INDEX`

Un nomi de type index était attendu, par exemple :

```
var i
w=sum(i=1,2 of i)
```

`nomi DE TYPE VAR`

Un nomi de type var était attendu, par exemple :

```
char v
forv v in (1/2,1/3)
```

`nomi INCONNU`

Dans la définition d'un nom par `access`, le nom accédé doit avoir un type. Par exemple, si `z` est un nom inconnu, on obtient cette erreur dans :

```
local dataa z access zp
```

`NON ENTIER`

L'argument doit être entier > 1, par exemple dans :

```
w=prfact(1/2)
```

ou

```
w=mdinv(2*x+1,x^2+2,1/3)
```

`NON NUMERIQUE`

Un nombre était attendu, par exemple au lieu de `kl` dans la commande éditeur (`pnt` étant un label) :

```
#pnt+kl
```

`NON RATIONNEL`

Dans :

```
print numr(1~)
```

l'argument ne doit pas être flottant. Dans :

```
complex i
print cxabs(1+i)
```

le résultat n'est pas rationnel.

`NUMERO D'ENREGISTREMENT`

En désignant par L le numéro du dernier enregistrement d'un fichier en accès sélectif, le numéro d'enregistrement doit être un entier de 1 à L (lecture) ou de 1 à $L+1$ (écriture).

`OVERFLOW`

Nombre trop grand, par exemple :

```
w=2^(2^14)+1
w=w^4
```

`PAS DE PLACE DISQUETTE`

Lors d'une écriture d'un fichier ou programme sur disque.

`PAS DE PROGRAMME`

Dans le menu Help, lorsqu'on demande le transfert d'un programme vers la source (F20 par exemple), alors qu'il n'y a pas de programme sur la page.

`PERMUTATION`

Une permutation des N premiers entiers était attendue, par exemple dans :

```
index p(10)
ift nextperm(10,p(1))
```

`PILE PROC DETRUITE`

Lors d'un retour de procédure, une anomalie de la pile est détectée. C'est une erreur de logique interne du Basic, qui n'apparaît pas normalement.

`PILE VIDE`

Lorsque plus de `pop` et `pop$` que de `push` et `push$` sont effectués, par exemple :

```
x=pop
```

Peut se produire, avec en prime des mauvais dépilages, si on clique sur `pop` dans le débogueur.

`PLANTAGE EN VUE. REINITIALISER`

Erreur système lors des réallocations mémoire par `himem`.

`POINTEUR MEM:`

Cette erreur peut se produire si un fichier virtuel "MEM:" est mal manipulé en tant que variable, comme dans l'exemple :

```
c$=space$(100)
open "a",1,"mem:",c$
c$=
print #1,"a"
```

où l'écriture qui devrait se faire en 100 (canal de type "a"), n'est plus possible après vidage de `c$`.

`POLYNOME`

Un polynôme était attendu. Une autre cause est le calcul d'expressions non polynomiales en présence d'une condition entière, comme dans :

```
cond 17
w=(1/y+x)^17
```

`RACINE`

Le degré de la racine doit être en entier positif. Erreur dans :

```
print root(x,-1)
```

`READ/DATA`

Il n'y a pas assez de `data` pour les commandes `read`.

`REMEMBER INTERDIT OU REPETE`

La commande `remember` ne peut se trouver que dans une fonction, et il ne peut y en avoir plusieurs exécutées dans le même appel.

`REPERTOIRE`

Erreur lors de la donnée d'un répertoire, dans `chdir` par exemple.

`RETURN SANS APPEL`

La commande `return` a été trouvée au niveau 0 de sous-programme. Peut-être manque-t-il une commande `stop` à la fin du programme principal, ce qui fait que l'exécution a continué dans un sous-programme.

`R_FILES TROP PETIT`

Il y a trop de fichiers de type "R". Il suffit d'augmenter la valeur de la variable d'état `r_files`.

`SIZE`

La taille d'un index est incorrecte. Par exemple (pas de taille 7) :

```
index*7 sept
```

ou (la taille doit être 8, 16 ou 32 dans `copy`) :

```
index*1 g(10)
copy g(0),3,1,g(1),1
```

`SORTIE IMPRIMANTE`

L'imprimante n'est pas branchée.

`SOURCE VIDE`

Essai de sauvegarder une source vide (commande de l'éditeur).

`SUBSTITUTION INCOMPLETE`

La valeur du littéral `y` manque dans la substitution flottante :

```
w=fsubs(x*y+1,x=1~)
```

`S_COND TROP PETIT`

Trop grand nombre de conditions. Il suffit d'augmenter la valeur de la variable d'état `s_cond`.

`S_MENU TROP PETIT`

La place réservée pour l'arbre du menu est insuffisante. Il suffit d'augmenter la valeur de la variable d'état `s_menu`.

`S_NAME TROP PETIT`

Erreur rare qui ne peut survenir que si une commande utilise un très grand nombre de noms non encore déclarés. Il suffit d'augmenter la valeur de la variable d'état `s_name`.

`S_PRO TROP PETIT`

Beaucoup d'appels imbriqués de sous-programmes ou de boucles. Cette erreur peut être une erreur de programmation (boucle d'appels de procédures sans retours, etc.), comme dans :

`sansfin:sansfin`

S'il y a vraiment besoin d'un grand degré d'imbrication, il suffit d'augmenter la valeur de la variable d'état `s_pro`.

`S_VAR TROP PETIT`

Le nombre de variables internes est insuffisant, par exemple dans un calcul de déterminant d'ordre n, qui en a besoin de n^2. Il suffit d'augmenter la valeur de la variable d'état `s_var`.

`S_XQT TROP PETIT`

Trop de noms locaux ou d'appels par nom (`@n`). Cette erreur peut être une erreur de programmation, comme pour l'erreur de `s_pro` ci-dessus. Si le programme est correct, il suffit d'augmenter la valeur de la variable d'état `s_xqt`.

`TROP DE DONNEES`

Par exemple (la deuxième donnée n'est pas utilisée) :

```
local datav 1,2 var v
```

`TROP DE LIT/VAR`

Le nombre de littéraux, ou de variables est limité à 2^{15}, en comptant tous les éléments des tableaux. Par contre le nombre d'index n'est pas limité.

`TYPE ILLEGAL`

Diagnostic un peu moins précis que `nomi de TYPE` ..., par exemple pour les types mélangés dans

```
var v
index i
exg i,v
```

`VALEUR APRES @`

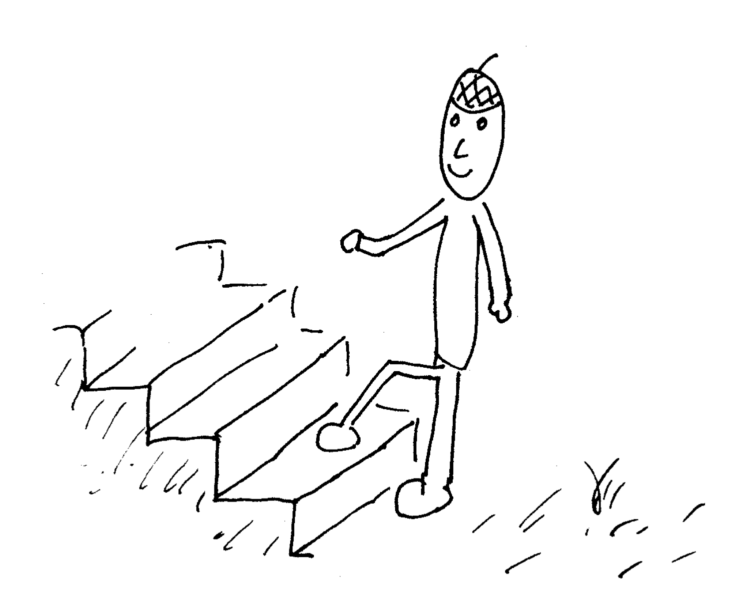
Autre qu'un entier de 0 au nombre d'arguments du sous-programme.

`VALEUR TROP FAIBLE`

Lors de la modification des variables d'état, Basic 1000d exige une valeur minimum.

9

Notions de base



Les noms

nom

Les noms représentent des objets manipulés par le Basic 1000d. Ils doivent s'écrire sans espaces avec les lettres A–Z, a–z, les chiffres 0–9 et les 7 symboles "!#$%.?_". Le premier caractère doit être une lettre. Après le nom on doit trouver soit un espace, soit un caractère autre que A–Z, a–z, 0–9 ou "!#$%.?_". La longueur des noms doit être au plus 32 caractères. Chaque nom ne peut décrire qu'un objet en Basic 1000d. Ainsi vous ne pouvez pas utiliser une variable `loop` :

```
loop=28
```

parce que `loop` est une commande du Basic 1000d.

Les noms peuvent être indicés, sans que le nombre d'indices dépasse 15. Les indices doivent avoir une valeur entière ≥ 0.

Définitions du manuel

Nous introduisons ci-dessous la définition de "nomi" pour décrire les noms indicés de façon formelle. Nous utilisons partout la même présentation. Les divers cas possibles sont séparés par "|". Les ";" signalent des commentaires. Des exemples suivent la définition. Souvent, la définition fait appel à d'autres définitions (ici à nom et ind), auxquelles on se référera aisément grâce à l'index.

nomi

```
  nom(ind {,ind})
| nom
```

La première forme est utilisée pour les tableaux (de types char, var, index, lit). La définition précise de "ind" est reportée à plus tard.

Exemple de nomi

Les formes `c$` et `Mois$(i+3)` sont aussi des nomi.

```
dim Mois$(12),Semaine$(7),Tablette(4,4,70)
c$=Mois$(i+3)
```

Espaces

Les espaces sont interdits dans les noms et devant "(" dans les nomi et fonctions. Ainsi :

```
w=sin (1)
```

qui provoque une erreur Arguments De Fonction, doit être récrit sans espace. Parfois des espaces sont nécessaires comme après x dans :

```
ift x print 1
```

Ils sont très souvent ignorés comme dans :

```
x  = intlg( 1  +  4  )  *  5
```

Type

A chaque nom est associé un type. Le type d'un nom peut être obtenu par la commande suivante.

TYPE nom {, nom}

Commande Ecrit le type des noms

La commande `type` donne aussi des indications sur les arguments dans le cas d'une fonction. Elle est utile pour la mise au point des programmes, car beaucoup d'erreurs de syntaxe sont dues à l'utilisation de mauvais types.

Exemple

```
     type print,int,chr$
```

Sortie (145 ms)

```
print est de type commande
int est de type V_fonction(réel[, ])
chr$ est de type C_fonction( [, ])
```

Table des types

type	exemple	
Commande	`print`	
V_fonction	`int`	Fonction à valeur exacte
	`sin`	Fonction à valeur flottante
C_fonction	`chr$`	Fonction chaîne de caractères
Constante	`ten`	Valeur numérique non modifiable
Variable d'état	`base`	
Lit (littéral)		
Var		
Index		
Char		
Label/proc		
Label de V_fonction		
Label de C_fonction		

Les types sans exemples sont des types utilisateurs, les autres sont les types des mots clefs internes.

Mots clefs

Certains noms sont les mots clefs du Basic 1000d, par exemple `print`. Nous venons de voir le mot clef `type`, ainsi que sa syntaxe. Nous présentons tous les mots clefs de façon analogue. La syntaxe est résumée dans le titre. Parfois plusieurs mots clefs sont regroupés, pour une présentation commune. Après le(s) titre(s), on trouve le type du mot clef (Commande, Variable d'état, etc.) et une description sommaire. Suivent la description détaillée et les exemples.

Les mots clefs peuvent s'écrire indifféremment en majuscules ou minuscules par exemple `print`, `PrinT`, etc. Les autres noms, particuliers à votre programme, par contre distinguent ou non les majuscules des minuscules. Tout dépend des mots clefs suivants :

DISTINGO [k]

NODISTINGO [k]

Variables d'état Contrôlent la distinction majuscules/minuscules

k

entier*16

Les mots clefs `distingo` et `nodistingo` peuvent s'utiliser comme commande, pour activer l'option distingo ou nodistingo, ou bien être lus comme des variables. Ils renvoient alors la valeur 0 (si distingo est actif) ou 1 (si nodistingo est actif). Il y a en Basic 1000d de nombreux mots clefs semblables, d'où la nécessité de la définition suivante.

Variable d'état (type de mot clef)

Les variables d'état peuvent apparaître en tant que V_fonction sans argument, ou bien en tant que commande qui modifie la valeur de cette V_fonction. Elles s'utilisent comme des variables ordinaires, mis à part que l'assignation d'une valeur se fait sans signe égal, par exemple :

```
base $A
```

et non :

```
base=$A
```

La présence du signe égal n'a pas été autorisée pour éviter que par erreur on donne un nom de variable d'état à une variable ordinaire du programme. Par exemple, si vous ignorez encore que `date` est une variable d'état, il se peut que vous utilisiez une variable `date` :

```
date=1515
```

qui sort en erreur pour vous aviser de la qualité de `date`. Cependant, le signe d'assignation "=" a été autorisé, d'une part dans le cas des variables `date$` et `time$`, pour compatibilité avec d'autres Basics, et d'autre part dans le cas des variables d'état indicées comme `color`(i).

Après cette digression sur les variables d'état, revenons à `(no)distingo`. En pratique, si on veut que les noms distinguent tout le temps les minuscules des majuscules, on oubliera ces variables d'état. Si par contre on ne veut jamais distinguer les minuscules des majuscules, on placera en première ligne de la source et de la bibliothèque :

```
nodistingo
```

Exercice Nb_nom

Quel est le nombre de noms possibles en Basic 1000d?

Le reste de cette section, très délicat, doit impérativement être sauté en première lecture. Les commandes `distingo` et `nodistingo` agissent également

sur les comparaisons de chaînes, et les fonctions comme `instr`, `decode` et `cmp`. Notons que seulement les 26 lettres non accentuées sont concernées (pour les noms, les lettres accentuées sont interdites). Les deux commandes :

```
nodistingo
nodistingo u
```

où `u` a pour valeur un entier impair (1 par exemple), sont équivalentes. Elles agissent sur les labels en dessous de la ligne de la commande, et sur les noms définis après l'exécution de la commande, même si ces définitions ont lieu sur une ligne située avant la commande dans la source. Les majuscules et minuscules sont alors équivalentes. Elles n'ont aucun effet sur les noms définis avant la commande.

Exemple

La variable `w`, définie avec l'option distingo, ne peut être référencée par `W`, même après passage en option nodistingo. L'impression de `W`, qui est de type inconnu, provoque la création d'un littéral `w` (en option nodistingo, les nouveaux noms des littéraux sont écrits en minuscules par le Basic). Il en résulte que l'ancienne variable `w` devient inaccessible.

```
var w
nodistingo
type w,W
print w;W
type w,W
```

Sortie (170 ms)

```
w est de type var
W est de type inconnu
  0  w
w est de type lit
W est de type lit
```

La commande :

```
nodistingo p
```

où `p` a pour valeur un entier pair (0 par exemple), agit sur les labels en dessous de la ligne de la commande, de sorte que les majuscules et minuscules sont équivalentes. Elle agit de façon opposée sur les noms définis après l'exécution de la commande, en distinguant les majuscules des minuscules. En effet, d'une part, le nom des labels est déterminé par un premier passage dans la source, sans exécution. Ce passage reconnaît les commandes `(no)distingo`, mais ne calcule pas l'argument. D'autre part, Basic 1000d est un langage interprété dans lequel le nom d'une variable ou index, n'est enregistré que lorsqu'il apparaît lors de l'exécution. C'est la valeur de la variable d'état `(no)distingo` à ce moment là qui est utilisée.

La commande :

```
xqt "nodistingo"
```

n'agit pas sur les labels. En effet elle est cachée, et non vue lors de l'enregistrement des labels.

La commande :

```
nodistingo distingo
```

au contraire de la précédente, agit sur les labels, mais pas sur les noms de variables. En effet, elle assigne à la variable d'état `(no)distingo` sa propre valeur, ce qui ne la modifie pas.

Exemple

Le nom `WAB` est défini alors que nodistingo est actif. Tous les noms `WAB`, `WAb`, `WaB`, etc. désignent le même objet. Par contre les noms `uab` et `Uab` et `definir` ont été définis avec distingo actif. Ce sont donc des objets différents, et les noms `UAB` et `Definir` sont inconnus du Basic. Remarquer que les deux noms de la même ligne, `definir` et `WAB` sont traités de façon différente. Le label `definir` est trouvé avec distingo actif lors de la lecture des labels. La variable `WAB` est trouvée avec nodistingo actif lors de l'appel de la procédure `definir`.

```
        nodistingo
        print distingo;nodistingo
        definir
        distingo
        print distingo;nodistingo
        uab=x^9
        Uab=y^3
        print WAB;wab;uab;Uab
        type UAB,Definir
        stop
definir:WAB=1/8
        return
```

Sortie (250 ms)

```
  1  1
  0  0
  1/8  1/8  x^9  y^3
UAB est de type inconnu
Definir est de type inconnu
```

Les nombres exacts

Un nombre exact est représenté par un entier m ou une fraction m/n avec m et n premiers entre eux. Tous les nombres avec m et n inférieurs en valeur absolue à 2^{65520} (approximativement 10^{19723}) peuvent être représentés.

BASE k

Variable d'état Base numérique

k

entier*16 $k \in [2, 36]$

La valeur de la base (dix, après `clear`) utilisée pour l'entrée et la sortie des nombres peut être modifiée par l'instruction :

```
base k
```

Les chiffres au delà de 9 sont les lettres en majuscules ou minuscules A, B, ..., Z. Le mot clef `base` est une variable d'état. On peut modifier sa valeur par l'instruction ci-dessus, ou bien on peut le faire entrer dans une expression.

Exemple

Le mot clef `base` est d'abord utilisé en tant que commande, avec la valeur 12, pour changer la base des nombres. Il est ensuite utilisé en tant que fonction, pour calculer x, puis pour sortir sa valeur. Remarquer l'option D du `print` qui permet d'écrire `base` en décimal. Sinon l'impression a lieu dans la base `base`, et s'écrit toujours 10.

```
base 12
x=base^2
print /d/base;x
```

Sortie (25 ms)

```
12  144
```

Exemple

Conversion de base 10 en base 3 et inversement Le symbole § indique une valeur en base 10. Le programme montre que :

$$(10000)_{10} = (111201101)_3$$
$$(0.01)_3 = \frac{1}{(9)_{10}}.$$

```
base 3
print §10000
w=.01
base §10
print w
```

Sortie (45 ms)

```
111201101
1/9
```

Constante (type de mot clef)

Les constantes s'utilisent comme des V_fonctions sans arguments. Les constantes suivantes correspondent aux valeurs courantes de `base`.

TWO

EIGHT

TEN

SIXTEEN

Constantes

Les constantes `two`, `eight`, `ten` et `sixteen` renvoient respectivement les valeurs 2, 8, 10 et 16.

Exemple

Pour revenir en base dix on peut utiliser une quelconque des commandes

```
base ten
base §10
base $A
base %1010
```

mais pas `base 10`, qui est sans effet.

Entrée des nombres

En plus des chiffres, on dispose des symboles `+ - . E $ § %` et `|`

Exemples

`.003` nombre en notation décimale.
`0A000` en base > dix, si le premier chiffre est une lettre, il faut le faire précéder de 0, sinon le Basic croit lire un nom.
`16 E-12` notation exponentielle de $16\ 10^{-12}$.
`10.3 E+10` et `10.3|10` valent $103\ 10^{9}$.

Lorsque `E` est aussi un chiffre (en base 16 par exemple) un espace est nécessaire devant le `E` représentant l'exposant. L'exposant peut aussi s'indiquer par `e` minuscule ou `|`.

Exemple

```
base 16
print 1e10;1 E10;1|10
```

Sortie (55 ms)

```
1E10  10000000000000000  10000000000000000
```

Si on fait précéder le nombre de `$` (resp `%`, §) le nombre est décodé en base 16 (resp 2, 10). Attention, dans `$A.A/15` seul `A.A` est en base 16, 15 est en base courante. La raison est que dans p/q (comme dans $p*q$, $p+q$, ...) le Basic décode 2 nombres et effectue une opération.

Calculs en nombres exacts

Les calculs rationnels sont faits sans approximation et donnent donc des résultats exacts.

Exemple

Les deux nombres rationnels a et b sont traités exactement, ce qui donne la valeur exacte $a-b=1$. Ces deux nombres sont de l'ordre de 10^{297}. La fonction `mlen` permet de voir que a occupe 198 octets en mémoire. L'écriture de la valeur exacte de a prendrait 6 lignes de 80 caractères.

```
        a=1+12^345/7^89
        b=4^345*3^345/7^89
        print mlen(a)
        print float(a)
        print a-b
```

Sortie (615 ms)

```
  198
  0.1270001767~ E+298
  1
```

Exercice Grand

Ecrire exactement le plus grand entier accepté par Basic 1000d. C'est $2^{65520} - 1$, mais attention, l'exponentiation exacte a^b est limitée à $b < 2^{15}$.

Temps des calculs en exacts

Voici les temps (en s) de quelques multiplications et divisions exactes

p	q	$p * q$	p/q
10^{100}	10^4	.001	.006
10^{100}	10^5	.003	.017
10^{100}	10^{50}	.006	.02
10^{1000}	10^4	.005	.02
10^{1000}	10^5	.01	.11
10^{1000}	10^{50}	.02	.17
10^{1000}	10^{500}	.14	.46
10^{5000}	10^{50}	.09	.8

puis les temps du calcul de 10^k et de sa racine carrée exacte par `intsqr`.

k	10^k	`intsqr`
100	.010	.025
500	.045	.175
1000	.13	.56
2000	.49	1.96
5000	2	15
10000	10	59
19000	37	148

Les temps de calculs de nombres terminés par beaucoup de 0 en binaire sont plus courts (0.01 s pour calculer `2^16000*2^16000` ou `4^16000`).

Les nombres flottants réels

Basic 1000d accepte des nombres approchés avec une précision limitée, ce sont les nombres flottants. Cette forme est avantageuse du point de vue temps et place mémoire dans les calculs numériques. Par contre les calculs algébriques, qui doivent être faits de façon exacte, n'acceptent pas l'utilisation directe de nombres flottants.

Entrée

Il suffit de rajouter un tilde après le nombre pour indiquer qu'il doit être considéré comme nombre flottant, par exemple :

```
7.345~    7.345 E+5~    7.345~ E+5    7.345~ E+5~
```

Les valeurs permises pour un nombre flottant réel, outre zéro, vont en valeur absolue environ de $1/N$ à N où $N \approx 10^{9850}$ (N dépend de la précision). Si une opération a un résultat inférieur à $1/N$ en module, le résultat est mis à zéro.

```
print 10~^-10000
print 10~^-9000
print 10~^9000
print 10~^10000
```

Sortie (405 ms)

```
 0~
 1.0000000000~ E-9000
 1.0000000000~ E+9000
*ERREUR* DOMAINE DE DEFINITION
print 10~^10000?
```

Remarquer que Basic 1000d accepte 10^{10000} en exact mais pas en flottant.

Les nombres complexes

Basic 1000d peut calculer en nombres complexes, une fois que le littéral complexe, représentant $\sqrt{-1}$, ait été défini par la commande `complex`. Cette commande, et les particularités des calculs en expressions complexes seront étudiées dans un chapitre ultérieur.

Exemple

Nous donnons ici seulement quelques exemples de nombres complexes. Une expression complexe sera exacte, sauf si elle contient un nombre flottant réel. Ainsi, `w` est initialisé avec un nombre flottant complexe, et `x` avec un nombre complexe exact.

```
complex i
w=1/7*i+2/3~
x=1/3+i/4
y=float(2,3)
z=float(x)
print w
print x
print y
print z
```

Sortie (240 ms)

```
0.6666666667~ +i*0.1428571429~
1/4*i +1/3
0.2000000000~ E+1 +i*0.3000000000~ E+1
0.3333333333~ +i*0.2500000000~
```

Sortie des nombres

Les nombres réels, qu'ils soient exacts ou flottants, peuvent sortir dans divers formats. On peut les sortir sous la forme d'un nombre fractionnaire m/n, où m et n sont des entiers premiers entre-eux. Cette forme représente exactement le codage interne du nombre. Pour un nombre flottant, le dénominateur n est une puissance de 2. On peut sortir les nombres sous forme décimale fixe. Nous écrivons "décimal", mais ce mot doit être changé en "binaire", "octal", etc. suivant la valeur de la base. On peut sortir les nombres sous forme exponentielle, en imposant que l'exposant soit un multiple d'un entier m donné, et en précisant le nombre de chiffres devant et après le point décimal.

Le format d'écriture est réglé par les quatre variables d'état `format`, `formatx`, `formatl` et `formatm`. On dispose de possibilités de formatages supplémentaires, par `print using`, qui seront étudiées dans un chapitre ultérieur, comme par exemple l'insertion d'espaces et/ou virgules pour faciliter la lecture de grands nombres.

TILDE [k]

NOTILDE [k]

Variables d'état Présence ou Absence de ∼

k

entier*16

Dans la sortie des nombres, le symbole tilde ∼ indique que la forme écrite n'est qu'une approximation du nombre. L'écriture d'un nombre rationnel exact sous forme décimale est ainsi en général suivie de ∼, et toutes les écritures d'un nombre flottant, même sous la forme m/n, comportent un ∼ pour rappeler que la représentation est approchée. Il est cependant possible de supprimer l'écriture de ∼, en effectuant la commande `notilde`.

Exemple

```
notilde
print pi
```

Sortie (65 ms)

```
0.3141592654 E+1
```

L'écriture de ∼ est rétablie après `clear` et après `tilde`. Les variables d'état `tilde` et `notilde` renvoient la même valeur, 0 quand ∼ s'écrit et 1 sinon. Lorsqu'on effectue une des commandes :

```
tilde k
notilde k
```

l'effet ne dépend plus que de la valeur de k. Si $k = 0$ (ou pair), le tilde s'écrit. Si $k = 1$ (ou impair), le tilde ne s'écrit pas.

Le Basic 1000d possède un grand nombre de variables d'état qui fonctionnent de façon analogue à la paire `tilde/notilde`. L'exemple suivant montre l'intérêt de ce type de fonctionnement.

Exemple

`NT` est une procédure qui écrit son argument `x` sans tilde, mais qui ne modifie pas l'option `tilde` ou `notilde` du programme appelant. Pour cela, la valeur d'entrée de `tilde` est poussée dans la pile par `push`. Avant le retour de la procédure par `return`, la valeur d'entrée de `tilde` est ôtée de la pile par `pop` et remise dans `tilde`.

```
   tilde random(2)
   print 1~
   NT(1~)
   print 1~
   stop
NT:procedure(x)
   push tilde
   notilde
   print x
   tilde pop
   return
```

Sortie (100 ms)

```
0.1000000000~ E+1
0.1000000000 E+1
0.1000000000~ E+1
```

FORMATX k

Variable d'état Format d'écriture des nombres exacts

k

entier*16

On peut choisir le format d'écriture des nombres exacts par la commande :

```
formatx k
```

où k est une expression prenant une valeur entière dans $[-2^{15}, 2^{15}[$. Si $k = 0$, la sortie se fait sous la forme exacte m/n. Si $k > 0$, la sortie se fait sous forme fixe avec $k-1$ chiffres après le point décimal. Tous les chiffres écrits sont exacts, mais s'ils ne suffisent pas pour représenter la valeur exacte du nombre, le dernier chiffre est arrondi et suivi de $\sim$. Si $k < 0$, la sortie est sous forme exponentielle `a E+b` où `a` est donné avec $|k| - 1$ chiffres après le point. Dans ce cas, le format dépend de plus des variables d'état `formatl` et `formatm`. Le mot clef `formatx` est une variable d'état et peut être utilisé dans des expressions.

Exemple

```
print "FORMATX"
for i=-15,15,5
  formatx 0
  print justr$(i,5);
  formatx i
  print justr$(formatx,27);justr$(1000/7,30)
next i
```

Sortie (650 ms)

```
FORMATX
  -15     -0.15000000000000 E+2        0.14285714285714~ E+3
  -10           -0.100000000 E+2              0.142857143~ E+3
   -5                 -0.5000 E+1                    0.1429~ E+3
    0                           0                         1000/7
    5                      5.0000                      142.8571~
   10                10.000000000                  142.857142857~
   15            15.00000000000000              142.85714285714286~
```

FORMAT k

Variable d'état Format d'écriture des nombres flottants

k

entier*16

Cette commande est l'analogue de `formatx k` pour les nombres flottants. En impression les nombres flottants sont suivis de $\sim$, sauf après `notilde`. Si $k = 0$,

la sortie se fait sous la forme m/n. Si $k \neq 0$, la sortie se fait sous forme fixe ou exponentielle avec $|k| - 1$ chiffres après le point décimal. Le mot clef `format`, comme `formatx` est une variable d'état et peut être utilisé dans des expressions.

Exemple

```
format 0
print 3/7~
```

Sortie (55 ms)

```
965057063007963/2251799813685248~
```

Le nombre flottant 3/7∼ est représenté en mémoire par un entier multiplié par une puissance de 2. C'est cette forme qui est écrite (le dénominateur est 2^{51}).

FORMATL l

FORMATM m

Variables d'état Format exponentiel

l

entier*16

m

entier*16 > 0

Le format exponentiel correspond à la forme $x = ab^k$ où b désigne la base et où k est un nombre entier. Le nombre x est alors écrit `ap E k` où le nombre `ap` est une valeur approchée de a. Il y a une infinité de telles écritures, suivant la valeur de k. Après les commandes :

```
format f  (ou formatx f)
formatl l
formatm m
```

où f (tel que $p = -f - 1 \geq 0$), l et m sont des entiers, l'écriture est déterminée de la façon suivante. L'exposant $k = hm$ est le multiple de m tel que :

$$b^{l-1} \leq a < b^{l+m-1}.$$

Le nombre écrit, `ap`, est le nombre à virgule fixe et à p chiffres après la virgule le plus proche de a.

Exemple

Le programme suivant écrit π et 10π pour diverses valeurs de f, l et m. La valeur de f est ici fixée par `using` et non par `format` f. Dernière ligne de la sortie, `0.~ E+1` indique que la valeur est plus près de 0 que de 10^1.

```
print "  f  l  m       pi                          10*pi"
FR 10,-1,1
FR 10,0,1
FR 10,1,1
FR 10,2,1
FR 10,-1,2
FR 10,0,2
```

```
   FR 10,1,2
   FR 10,2,2
   FR 0,1,3
   FR 0,0,1
   stop
FR:procedure(I,l,m)
   FORMATL l
   FORMATM m
   C$="#####."&STRING$(I,"#")&"~"&SPACE$(11-I)&"^^^^^"
   print justr$(-I-1,3);justr$(l,3);justr$(m,3);
   print USING C$,PI,PI*10
   return
```

Sortie (1295 ms)

```
  f  l  m      pi                   10*pi
-11 -1  1    0.0314159265~  E+2     0.0314159265~  E+3
-11  0  1    0.3141592654~  E+1     0.3141592654~  E+2
-11  1  1    3.1415926536~          3.1415926536~  E+1
-11  2  1   31.4159265359~  E-1    31.4159265359~
-11 -1  2    0.0314159265~  E+2     0.3141592654~  E+2
-11  0  2    3.1415926536~          0.3141592654~  E+2
-11  1  2    3.1415926536~         31.4159265359~
-11  2  2  314.1592653590~  E-2    31.4159265359~
 -1  1  3    3.~                   31.~
 -1  0  1    0.~            E+1     0.~            E+2
```

Sortie (3140 ms)

Remarques

Les valeurs $l = 0$, $m = 1$, $f = -11$, $p = 10$ sont les valeurs après `clear`. Dans ce cas la valeur écrite `ap` va de `0.1000000000` à `1.0000000000`. La valeur 1 est possible pour `ap` après arrondi, bien que strictement $a < 1$.

Calculs en flottant

PRECISION k

Commande Contrôle la précision des calculs en flottants

k

entier*16 $k > 0$

La commande `precision` k indique que les calculs en nombres flottants doivent être faits avec k chiffres significatifs (dans la base courante). On doit avoir

$b^{-k} > 2^{-4096}$ où b désigne la base. Ainsi en base $b = 10$, k peut être aussi grand que 1230. Après `clear` la précision est de 10 chiffres. De plus une commande implicite `format` $-k-1$, fixant le format de sortie des nombres flottants, est effectuée.

Exemple

On calcule le nombre flottant 1000/7~ avec la précision 25 et on l'écrit en format −26 (valeur implicite après `precision` 25) puis en format 51. En format 51 seuls les 25 premiers chiffres sont garantis exacts, parce que la représentation interne du nombre est prévue pour 25 chiffres seulement. On le vérifie en comparant avec le nombre exact 1000/7 qui sort avec tous ses chiffres exacts (sauf arrondi).

```
precision 25
print 1000/7~
format 51
print 1000/7~
formatx 51
print 1000/7
```

Sortie (220 ms)

```
0.1428571428571428571428571~ E+3
142.85714285714285714285714285713203847901411183809721~
142.85714285714285714285714285714285714285714285714286~
```

PRECISION2 k

Variable d'état Contrôle la précision des calculs en flottants

k

entier $k > 0$

La commande `precision2` k fixe la précision sans modifier `format`. La valeur k indique le nombre de bits significatifs (nombre de chiffres en base deux). Autrement dit, la commande équivaut à :

```
push base,format
base §2
precision k
format pop
base pop
```

Inversement `precision 100` en base 10 équivaut à :

```
precision2 342
format -101
```

Remarquer que `precision2` est une variable d'état mais que `precision` est seulement une commande.

Calculs en flottant

Les opérations + − ∗ / et ^, dans lesquelles apparaît un nombre flottant sont effectuées en flottant, seulement à la précision en cours. Une exponentiation

comme 2^(1/2) qui ne peut être calculée en exact est également calculée en flottant. Les V_fonctions flottantes sont calculées avec la précision courante.

Temps de calcul typiques en secondes

precision	10	11	100	200	500	1000	1230
$w*w$	0.002	0.002	0.01	0.03	0.1	0.6	0.9
$1/(w+1)$	0.01	0.01	0.03	0.07	0.3	0.8	1.2
w^{1000}	0.02	0.02	0.14	0.44	2	9	13
$\sqrt{w}$	0.01	0.01	0.05	0.11	0.4	1.3	2.8
e^w	0.03	0.06	0.89	3.1	18	85	135
$\log w$	0.05	0.09	1.38	4.7	27	124	258
w^w	0.07	0.14	2.21	7.7	46	210	396
$\sin w$	0.04	0.09	1.67	6.0	39	187	362
$\tan w$	0.04	0.18	3.30	12.1	78	374	726
$\arctan w$	0.05	0.12	2.95	11.6	78	396	880
print w	0.03	0.03	0.14	0.30	1.0	2.9	4.1

Le nombre réel flottant $w = 17/19\sim$ a été utilisé. Les V_fonctions flottantes sont calculées par polynômes en mémoire jusqu'à la précision 10 et par des développements limités et relations fonctionnelles au delà. C'est pourquoi les temps augmentent fortement en précision 11. Les calculs en précision inférieure à 10 ne sont pas recommandés, le gain en temps étant de 30% au mieux.

Tests d'exactitude et de vitesse

Ces tests sont destinés à vérifier l'exactitude des calculs avec diverses valeurs de la précision, et à comparer les temps nécessaires. Ici aussi on observera le changement au passage de précision 10 à 11.

Lorsqu'on compare à d'autres langages, il apparaît que, quand les flottants sont codés sur 8 octets, la précision de ces langages, pour les tests effectués ici, correspond à une précision entre 10 et 12 du Basic 1000d. En ce qui concerne la vitesse, à exactitude égale, et toujours pour ces tests, le Basic 1000d est environ 2 à 20 fois plus lent que les meilleurs. La lenteur relative du Basic 1000d vient principalement du fait que le Basic 1000d permet une précision variable, ce qui nécessite une allocation dynamique de la mémoire.

Le test 1 consiste à itérer 100 fois, à partir de $x_1 = 1$, la relation $x_{i+1} = f(x_i)$ où $f(x)$, qui est calculé par un produit de fonctions trigonométriques, est en théorie égal à x.

Le test 2 (test de Savage) consiste à itérer 2499 fois, à partir de $x_1 = 1$, la relation $x_{i+1} = f(x_i)+1$ où $f(x)$ est aussi une expression théoriquement égale à x.

Le test 3, d'exactitude, consiste à itérer 20 racines carrées puis 20 carrés en partant de 2.

Dans les tests, l'erreur indiquée `err` est la différence entre le résultat numérique et la valeur théorique exacte, et les temps `t` sont en secondes.

Voici le programme effectuant les tests.

```
notilde
formatl 1
print spc(12);"Test 1";spc(13);"Test 2";spc(9);"Test 3"
print "precision   ";string$(2,"t   err"&spc(12));"err"
forv p in (1,[9,15,1],[20,60,10])
  precision p
  clear timer
  print justr$(p,3);
  x=1
  for i=1,100
    x=x*(tan(x)*(cos(x)/sin(x)))
  next i
  print justr$(timer,10);using "###.^^^^^^^",x-1;
  clear timer
  x=1
  for i=1,2499
    x=tan(atn(exp(log(sqr(x*x)))))+1~
  next i
  print justr$(timer,8);using "###.^^^^^^^",x-2500;
  x=2
  for i=1,20
    x=sqr(x)
  next i
  for i=1,20
    x=x^2
  next i
  print using "   ###.^^^^^^^",x-2
nextv
```

Sortie

```
             Test 1             Test 2        Test 3
 precision   t   err              t   err           err
   1          7  2. E-4        257 -2. E+3       -1.
   9          8  6. E-12       308 -5. E-5        5. E-7
  10         10  4. E-12       364 -4. E-6        2. E-8
  11         35 -3. E-14       944 -3. E-5        2. E-8
  12         39 -2. E-15      1063 -2. E-6        9. E-11
  13         41 -1. E-16      1112 -1. E-7       -2. E-11
  14         44 -1. E-17      1173 -1. E-8        1. E-11
  15         44 -2. E-18      1197 -2. E-9        7. E-13
  20         67 -6. E-24      1758 -1. E-14       1. E-17
  30        102 -2. E-34      2653 -8. E-25      -4. E-28
```

```
40       159 -1. E-44      4200 -5. E-35     -10. E-38
50       217 -1. E-55      5779 -1. E-45       3. E-48
60       276 -8. E-66      7233 -2. E-55       4. E-59
```

Index, lit, var et char (types de noms)

Exemple

Nous avons déjà vu des variables (de type var) et littéraux (type lit) dans l'introduction voici un autre exemple :

```
Z3=(A+B)^2-(A-B)^2
Z=Z3/4
print Z
```

Sortie (50 ms)

```
A*B
```

Nous avons dans cet exemple, des littéraux `A` et `B` et des variables `Z` et `Z3` définis de façon implicite. Basic 1000d a donné le type var aux noms inconnus trouvés à gauche du signe = et le type lit (littéral) chaque fois qu'il a trouvé un nom inconnu à droite de =.

index*size (définition)

C'est un objet prenant des valeurs entières signées codées sur *size* bits.

size

La table suivante donne les valeurs possibles d'un index suivant sa taille *size*, qui doit être un des entiers 1, 2, 4, 8, 16 ou 32.

size	valeurs
1	$[-1, 0]$
2	$[-2, 1]$
4	$[-8, 7]$
8	$[-128, 127]$
16	$[-32768, 32767]$
32	$[-2147483648, 2147483647]$

INDEX [*size] nomi{, nomi}

Commande Déclaration d'index

Par défaut $size = 32$. Un index occupe une adresse fixe en mémoire. Les index indicés occupent chacun *size* bits.

Exemple

Le tableau `logic` occupe 1000000/8 octets en mémoire.

```
index *1 logic(999999)
```

△ L'implantation en mémoire due à `index t(2,1)`, par exemple, est réalisée suivant l'ordre `t`(0,0), `t`(1,0), `t`(2,0), `t`(0,1), `t`(1,1), `t`(2,1). Comme `t` est un index*32, `t`(1,0) est implanté en `ptr`(`t`(0,0))+4. Pour un index*4 défini par `index*4 t(2,1)`, `t`(0,0) est implanté dans les bits 7–4 et `t`(1,0) dans les bits 3–0 de l'octet en `ptr`(`t`(0,0))=`ptr`(t(1,0)). Les index*size non indicés sont implantés en commençant également par le bit 7, mais quelle que soit la valeur de leur taille, 4 octets sont réservés (on ne gagne pas de mémoire en utilisant une taille inférieure à 32 bits).

Les index*32 sont définis de façon implicite par une commande de boucle `for`.

Exemple

```
for i=-3,10000
next i
W=sum(j=1,1000 of 1)
type i,j
```

Sortie (1 s)

```
i est de type index
j est de type index
```

Exemple

Les index*32 a% et b! sont également définis de façon implicite, et il en est de même pour tout nom se terminant par % ou !, qui apparaît à gauche d'une assignation. Par contre c%, qui apparaît dans une expression, est un littéral.

```
a%=1
b!=7
w=c%
type a%,b!,c%
```

Sortie (115 ms)

```
a% est de type index
b! est de type index
c% est de type lit
```

LIT nomi {, nomi}

Commande Déclaration de littéraux

Exemple

```
lit X,Y,Z(2,3)
print sum(i=0,2 of Z(i,i))
```

Sortie (55 ms)

```
Z(2,2) +Z(1,1) +Z(0,0)
```

définit les littéraux simples `X`, `Y` et les 3×4 littéraux `Z`(2,3), `Z`(1,3), `Z`(0,3), `Z`(2,2), ..., `Z`(0,0). L'ordre donné est l'ordre de numérotation des littéraux qui

fixe l'ordre dans lequel les littéraux s'impriment. Le nombre de littéraux est limité à 2^{15} (le littéral indicé `Z` compte pour 12).

Les littéraux n'ont pas de valeur. La déclaration des littéraux simples est facultative, celle des littéraux indicés est obligatoire.

CHAR nomi {, nomi}

VAR nomi {, nomi}

Commandes Déclaration de variables de types char et var

Exemple

```
char c,ch(49)
var A,B,C,T(7),U(8,3)
```

La première instruction définit la variable simple `c`, et les variables `ch`(i) où i va de 49 à 0 (ordre décroissant) de type char. La deuxième instruction définit les variables simples `A`, `B`, `C` et les variables `T`(i) où i va de 7 à 0 (ordre décroissant) et les 9×4 variables `U`(8,3), `U`(7,3), ..., `U`(0,3), `U`(8,2), ..., `U`(0,0).

Les variables (les deux types var et char sont traités ensemble) sont numérotées à l'intérieur du Basic par des entiers dans $[0, 2^{15}[$. Le nombre de variables ne peut donc dépasser 2^{15}. Les ordres indiqués ci-dessus correspondent à des numéros croissants.

Exemple

La variable de type char `c$` est définie de façon implicite, et il en est de même pour tout nom se terminant par $ et à gauche d'une assignation. Par contre, `d$` qui apparaît à droite d'une assignation est considéré comme littéral.

```
c$=d$
type c$,d$
```

Sortie (90 ms)

```
c$ est de type char
d$ est de type lit
```

La déclaration des variables simples est facultative. La déclaration des variables indicées est obligatoire.

Les variables, à la différence des index, sont mémorisées à une adresse qui peut changer pendant l'exécution d'un programme. La place occupée en mémoire par une variable donnée est également variable et peut être aussi grande que tout l'espace disponible pour le programme.

Les variables de type char prennent pour valeur une chaîne de caractères de longueur quelconque (de 0 à tout l'espace libre) et de contenu quelconque.

Exemple

Le caractère `chr$`(0) dans la variable c$ provoque un changement de ligne à l'impression.

```
char CHANT
c$="Dimanche" & chr$(0) &"Lundi"
print c$
```

```
CHANT="o36 t5"&conc$(i=1,10 of f&chr$($41+random(26))&
 justl$(random(2)+1))
print CHANT
music CHANT
```

Sortie (4465 ms)

```
Dimanche
Lundi
o36 t5 P2 X2 W1 U1 X2 I2 E1 A1 S1 D2
```

Les variables de type var prennent 4 sortes de valeurs, soit un polynôme, soit un produit de polynômes, soit un nombre flottant réel, soit un nombre flottant complexe.

Δ Le codage de la valeur de la variable commence par un mot (2 octets) indiquant la forme de son contenu. Pour un nombre flottant ce mot est égal à –1 (réel) ou –2 (complexe), pour un polynôme 0. Une autre valeur atteste de la forme factorisée (et donne le nombre de facteurs).

IMPLICIT INDEX rg {, rg}

IMPLICIT CHAR rg {, rg}

Commandes Types implicites des noms

rg

```
    a
| a-b
```

indique un ou plusieurs caractères

a, b

Caractères alphanumériques (ceux qui servent à écrire les noms)

Par exemple `I-L` définit l'ensemble des lettres `I`, `J`, `K`, `L` (et aussi en minuscules si `nodistingo` est actif).

Nous avons indiqué que lorsque Basic 1000d trouve un nom inconnu à gauche d'une assignation, le nom est mis de type var. La commande `implicit` permet de modifier cette règle.

Exemple

```
implicit index I-M
implicit char C
```

a pour effet que tous les noms inconnus, à gauche d'une assignation, qui se terminent par `C` seront pris de type char, et que ceux qui se terminent par une des lettres de `I` à `M` seront pris de type index*32. Ces noms doivent, bien sûr, toujours être sans indice, sinon on a l'erreur "dimension non définie". Si le nom est à droite d'une assignation, il est toujours pris de type littéral.

Après `clear`, on a seulement les types implicites :

```
implicit char $
implicit index %,!
```

Certaines commandes forcent le type d'un nom, et ce type peut différer du type implicite du nom.

Exemple

Des types inhabituels sont attribués à `I$` et `C$`.

```
var I$
for C$=1,8
next
type I$,C$
```

Sortie (80 ms)

```
I$ est de type var
C$ est de type index
```

DIM sn { , sn }

Commande Déclaration de tableaux d'index, char et var

sn

```
  nomi
| *size nomi
```

Le type donné à nomi par la commande `dim` correspond à son type implicite, ou au type index si `*size` (voir `index`) est donné. L'effet de la commande `dim` sur ce nom est identique à la commande `var`, `char` ou `index` correspondant au type.

Exemple

```
dim z$(2),y(9),*16 j(7)
type z$,y,j
```

Sortie (145 ms)

```
z$ est de type char (2)
y est de type var (9)
j est de type index (7)
```

Initialisation Redéclaration

Il est possible de déclarer plusieurs fois le même nom de type var, char, index ou lit, à condition que ce soit toujours le même type, et les mêmes dimensions. Lors de la première déclaration, les index et les variables de type var sont mis à zéro, et les variables de type char sont vidées. Lors des déclarations suivantes, le Basic vérifie seulement le type et les dimensions, mais les valeurs ne sont pas réinitialisées.

Accès unidimensionnel des tableaux

Rangeons les éléments d'un tableau en faisant d'abord croître le premier indice, puis le deuxième, ... Par exemple le tableau

```
var T(2,5)
```

correspond à l'ordre `T`(0,0), `T`(1,0), `T`(2,0), `T`(0,1), ..., `T`(2,5).

Δ Le fait que cet ordre soit décroissant pour les numéros des variables et littéraux, et croissant pour les adresses des index est ici sans importance.

Il est possible d'accéder à ces éléments à l'aide des mots clefs `min` et `max` par les écritures suivantes :

nom(MIN [sk])

nom(MAX [sk])

nom

nom de tableau (de type var, char, index ou lit)

sk

expr commençant par un signe + ou –, et à valeur entière. La définition de expr est reportée à plus tard.
Les $3 \times 6 = 18$ éléments du tableau `T` donné en exemple peuvent s'écrire `T(min)`, `T(min+1)`, `T(min+2)`, ..., `T(min+17)` ou encore `T(max-17)`, `T(max-16)`, ..., `T(max)` dans le même ordre que plus haut. L'écriture `T(1+min)` est incorrecte.

DVARNUM(nomi)

V_fonction Numéro d'ordre d'un élément de tableau.
Renvoie un entier dans $[0, N[$ où N est le nombre d'éléments du tableau. L'élément `T`(a, b) du tableau `T`, de numéro d'ordre k = `dvarnum(T`(a, b)`)` peut être référencé par `T(min+`k`)`.

Exemple

A l'aide de `dvarnum`, on calcule le nombre d'éléments d'un tableau, puis on montre comment l'accès unidimensionnel d'un des éléments du tableau peut être effectué.

```
var Z(3,4,7)
print "Le tableau Z a";dvarnum(Z(max))+1;" éléments"
a=random(4)
b=random(5)
c=random(8)
print using "Z(#_,#_,#) peut s'écrire Z(min _+###)", a,
  b, c,dvarnum(Z(a,b,c))
```

Sortie (125 ms)

```
Le tableau Z a  160 éléments
Z(3,1,1) peut s'écrire Z(min + 27)
```

Formes exactes

Polynômes

Exemple

On écrit des polynômes (A, B, X, Z sont des littéraux)

```
print A+B
print 3*Z+1/27+A^25*X
```

Sortie (60 ms)

```
A +B
A^25*X +3*Z +1/27
```

On peut utiliser jusqu'à 2^{16} monômes et 2^{15} littéraux par polynôme. Les coefficients d'un polynôme peuvent être des nombres rationnels quelconque (les flottants ne sont pas admis). Les exposants doivent être dans $[0, 2^{15}[$.

Produit de polynômes

C'est un produit de la forme :

$$P_1 \prod_{i=2}^{s} P_i^{k_i}$$

où s $(0 < s < 65534)$ est le nombre de facteurs, P_1 est un nombre rationnel, et pour chaque i de 2 à s (si $s \neq 1$) P_i est un polynôme et k_i un entier dans $[-2^{15}, 2^{15}[$. Les polynômes P_i sont normalisés.

Polynôme normalisé (définition)

Tous ses coefficients sont entiers, de pgcd =1 et le premier coefficient écrit est positif.

Δ Cette définition dépend (pour le signe) de l'ordre de définition des littéraux.

De plus, le pgcd de 2 polynômes P_i et P_j $(i \neq j)$ est 1, les facteurs x^a (où x est un littéral) sont tous apparents.

Exemple

C'est le Basic qui se charge de donner une forme standard à un produit. Ici, le facteur commun $a - b$ est dévoilé.

```
print (a^2-2*a*b+b^2)^-1*(2*a^3/c-2*b^3/c)^-1
```

Sortie (200 ms)

```
1/2* [c]* [a -b]^-3* [a^2 +a*b +b^2]^-1
```

Les facteurs sont toujours premiers deux à deux, mais pas nécessairement irréductibles, ce qui fait qu'une expression donnée peut avoir plusieurs formes produit de polynômes, par exemple :

```
print 1/(a^2+2*a*b+b^2)
print (a+b)^-2
```

Sortie (75 ms)

```
[a^2 +2*a*b +b^2]^-1
[a +b]^-2
```

Forme factorisée ou développée

Ce paragraphe ne concerne que les expressions exactes. Les calculs en flottant ne dépendent pas des commandes qui suivent.

DEVELOP [k]

FACTOR [k]

Variables d'état Contrôlent la forme des calculs et assignations

k

entier*16

Chacune des commandes :

```
develop
develop k
factor k
```

où `k` est pair a pour effet de mettre l'option `develop`. Les deux variables d'état prennent alors la valeur 0. Les commandes :

```
factor
develop k
factor k
```

où `k` est impair ont chacune pour effet de mettre l'option `factor`. Les deux variables d'état prennent alors la valeur 1.

Avec l'option `develop` (qui est l'option au lancement du programme ou après `clear`), les assignations sont faites si possible avec la forme polynôme. Avec l'option `factor`, toutes les assignations seront faites sous la forme produit de polynômes. Il est possible de changer plusieurs fois l'option au cours du programme.

Exemple

Même après `develop`, `W` reste sous forme produit, car ce n'est pas un polynôme. `V` est développé sous forme polynôme, mais son expression a du être calculée sous forme produit (l'explication sera vue un peu plus bas). Cela est apparent dans le dernier `print`, où l'expression est sortie sous forme produit, sans conversion en forme développée comme dans une assignation.

```
develop
W=A/(A-1)
print W
V=A/B*(B+B*A)^7
print V
print A/B*(B+B*A)^7
```

Sortie (395 ms)

```
 [A]* [A -1]^-1
A^8*B^6 +7*A^7*B^6 +21*A^6*B^6 +35*A^5*B^6 +35*A^4*B^6 +21*A^3*B^6
 +7*A^2*B^6 +A*B^6
 [B]^6* [A]* [A^7 +7*A^6 +21*A^5 +35*A^4 +35*A^3 +21*A^2 +7*A +1]
```

Exemple

Après assignation en mode `factor`, `Z` est mis sous forme produit. `V` qui est sous forme polynôme est sorti sans conversion.

```
V=(x+y)^3
factor
Z=V
print Za;V
```

Sortie (110 ms)

```
 [x^3 +3*x^2*y +3*x*y^2 +y^3]
x^3 +3*x^2*y +3*x*y^2 +y^3
```

Lors de calculs avec uniquement des nombres exacts, les nombres sont considérés soit comme des polynômes (réduits au terme constant), soit comme des produits de polynômes (réduits au facteur constant).

FORMD(p)

V_fonction Développe *p*

p

expr

La fonction `formd` renvoie un polynôme si possible. Sinon on obtient la forme produit de facteurs

$$P_1(\prod_{i=1}^{m} x_i^{e_i})\frac{N}{D}$$

où P_1 est un nombre rationnel, N et D sont des polynômes normalisés, les x_i sont des littéraux et les e_i des entiers.

Exemple

```
W=37/57*(1+x)^-3/(1+y)^-2*u^4/v
print W
print formd(W)
print num(W)
print den(W)
```

Sortie (360 ms)

```
37/57* [v]^-1* [u]^4* [y +1]^2* [x +1]^-3
37/57* [v]^-1* [u]^4* [y^2 +2*y +1]* [x^3 +3*x^2 +3*x +1]^-1
37/57*y^2*u^4 +74/57*y*u^4 +37/57*u^4
x^3*v +3*x^2*v +3*x*v +v
```

FORMF(p [, k])

V_fonction Factorise p

p

expr

k

entier*16 (par défaut $k = 1$)

La fonction `formf` factorise complètement tous les polynômes (ou formes partiellement factorisées), à un ou plusieurs littéraux, sur l'ensemble des nombres rationnels **Q**. C'est une des fonctions les plus utiles en calcul symbolique, en particulier pour la recherche des racines exactes d'une équation. C'est de loin la fonction du Basic dont le code est le plus compliqué et le plus long (environ 25000 octets). Comme le temps de calcul peut être assez grand, le paramètre k permet de réduire ce temps, en diminuant les possibilités de la fonction. Si $k < 0$ la recherche des facteurs est la plus rapide, et se limite à des réductions par la fonction `red` (cas multilittéral seulement). Si $k = 0$ en plus de la factorisation précédente, on trouve tous les facteurs multiples de p, par calcul du pgcd de p et d'une dérivée partielle. Si $k > 0$ la fonction renvoie p sous la forme du produit de facteurs irréductibles sur **Q**.

Exemple

La factorisation du polynôme `W` diffère suivant le paramètre k.

```
W=  X^6 +4*X^5*A -3*X^5*B -19*X^4*A^2 -12*X^4*A*B +3*X^
 4*B^2 +14*X^3*A^3 +57*X^3*A^2*B +12*X^3*A*B^2 -X^3*B^3
  -42*X^2*A^3*B -57*X^2*A^2*B^2 -4*X^2*A*B^3 +42*X*A^3*
 B^2 +19*X*A^2*B^3 -14*A^3*B^3
print "FORMF(W,-1)=";formf(W,-1)
print "FORMF(W,0)= ";formf(W,0)
print "FORMF(W)=   ";formf(W)
```

Sortie (1195 ms)

```
FORMF(W,-1)=   [X^3 -3*X^2*B +3*X*B^2 -B^3]* [X^3 +4*X^2*A -19*X*A^2 +
 14*A^3]
FORMF(W,0)=    [X -B]^3* [X^3 +4*X^2*A -19*X*A^2 +14*A^3]
FORMF(W)=      [X -B]^3* [X -2*A]* [X -A]* [X +7*A]
```

Exemple

Dans l'assignation `W=W` en mode `factor`, seulement les monômes x^α et un nombre rationnel sont mis en facteur. La multiplication par $2a-x$ permet de découvrir le facteur $2a - x$ de `W`. En effet dans une forme factorisée les facteurs sont toujours premiers deux à deux.

```
W=17/5*x^6 -68/5*x^5*a +68/5*x^4*a^2
factor
W=W
print W
print formf(W)
```

```
print W*(2*a-x)
```

Sortie (280 ms)

```
  17/5* [x]^4* [x^2 -4*x*a +4*a^2]
  17/5* [x]^4* [x -2*a]^2
 -17/5* [x]^4* [x -2*a]^3
```

Attention, si vous écrivez en mode développé `W=formf(W)` l'assignation redéveloppe la forme factorisée. Il faut donc utiliser le mode factorisé :

```
factor
W=formf(W)
```

Temps de calculs algébriques

Pour $w_1 = (1+x)^{128}$, $w_2 = (1+x)^{256}$, $w_3 = (1+x)^{512}$, $w_4 = x^{(50)}$ et $w_5 = x^{(200)}$ ($x^{(n)} = x(x-1)(x-2) \times \cdots \times (x-n+1)$ est le symbole de Pochhammer) sous forme développée nous indiquons la longueur du codage mémoire (pour l'écriture il faut environ 2.5 fois plus de caractères) et les temps de calculs (en s) de w_i, de la dérivée, de l'intégrale et de factorisation complète. Le "?" indique une erreur mémoire (1040ST).

	w_1	w_2	w_3	w_4	w_5
`mlen`	2060	7068	25916	1136	20432
calcul	3	22	205	0.8	29
`der`	1	6	32	0.2	6
`intg`	4	19	95	2	61
`formf`	4	15	65	9285	?

Calcul des expressions mathématiques

Nous allons définir de façon précise les expressions mathématiques. La forme la plus générale est notée expr. La définition fait intervenir un grand nombre de définitions, dont certaines (avec chaînes de caractères) ne seront vues que dans les sections suivantes. Pour simplifier nous n'avons pas fait apparaître la distinction entre exacts et flottants. Une opération entre exacts donne un

exact (mis à part l'exponentiation). Une comparaison donne un exact, les autres opérations donnent un flottant si un des opérandes est un flottant. Cependant, si ce résultat ne peut être converti en flottant comme dans `2~*x`, où `x` est un littéral, on obtient l'erreur Nombre Complexe. Indiquons ici sous quelle forme (factorisée ou développée) sont effectués les calculs d'expr.

Si l'option est `factor` tous les résultats des calculs intermédiaires sont sous forme factorisée.

Si l'option est `develop` :

a) Le résultat d'une exponentiation est polynomial si possible.

b) Le résultat de $A + B$, $A - B$, $A * B$ est sous forme polynôme si A et B sont tous deux sous forme polynôme. Sinon le résultat est factorisé.

c) A/B est calculé en factorisé et le résultat est converti en polynôme si c'est possible.

Exemple

Ecrit en factorisé, polynôme, factorisé, polynôme

```
print A/B; C*D/D; E/F*F; G/H*H/1
```

Sortie (125 ms)

```
[B]^-1* [A]  C   [ E]  G
```

expr

```
  exprn
| exprn { AND exprn }
| exprn { OR  exprn }
| exprn { XOR exprn }
| exprn { EQV exprn }
| exprn IMP exprn
```

exprn

```
  exprc
| NOT exprc
```

Les opérations logiques sont possibles uniquement sur des entiers*32 qui sont considérés comme mots longs. Noter la priorité des opérateurs logiques `xor`, `eqv`, `imp`, `and` et `or`, inférieure à celle de l'opérateur `not`, et l'exigence d'homogénéité (pas de `and` et `or` au même niveau). Voici les tables de vérité des opérateurs logiques. La valeur -1 correspond à "vrai" et 0 à "faux".

a	b	`not` b	a `and` b	a `or` b	a `xor` b	a `eqv` b	a `imp` b
-1	-1	0	-1	-1	0	-1	-1
-1	0	-1	0	-1	-1	0	0
0	-1		0	-1	-1	0	-1

Exemples

Les deux exprn suivantes sont identiques :

```
NOT a<7
NOT(a<7)
```

Les deux expr suivantes sont également identiques :

```
a AND NOT b AND c
a AND (NOT b) AND c
```

Dans l'expr :

```
(pride AND prejudice) OR (sense AND sensibility)
```

les parenthèses sont obligatoires. La forme suivante est illégale car 1/2 n'est pas entier*32 :

```
NOT 1/2
```

L'opérateur non associatif `imp` ne peut être répété comme les opérateurs associatifs `and`, `or`, `xor` et `eqv`. Ainsi

```
(a imp b) imp c
a imp (b imp c)
```

sont des expressions différentes où on ne peut supprimer les parenthèses.

Exemple

Le programme suivant sort les tables de vérité des opérateurs `and`, `or`, `xor`, `eqv` et `imp`.

```
      logic and
      logic or
      logic xor
      logic eqv
      logic imp
      stop
logic:print "  a  b  a @1f b"
      for a=-1,0
        for b=-1,0
          print a;b;"   ";a @1f b
      next b,a
      return
```

Sortie (835 ms)

```
  a  b  a and b
 -1 -1    -1
 -1  0     0
  0 -1     0
  0  0     0
...
```

exprc

```
  expra
| expra comparateur expra
| expra IN v_ensemble
```

```
  | expra NOT IN v_ensemble
  | exprchaîne comparateur exprchaîne
  | exprchaîne IN c_ensemble
  | exprchaîne NOT IN c_ensemble
```

Le résultat d'une comparaison de 2 expressions algébriques expra, ou de 2 chaînes de caractères exprchaîne, ou le résultat d'une relation d'appartenance, `in`, ou de non appartenance, `not in`, prend la valeur −1 (vrai) ou 0 (faux).

comparateur

```
    <
  | =<
  | <=
  | >
  | >=
  | =>
  | =
  | <>
```

Noter que =< et <= sont identiques ainsi que >= et =>. Les comparaisons d'égalité = et de non égalité <> peuvent être effectuées entre expressions quelconque, les autres seulement entre nombres réels ou exprchaînes. Le type entier*1 (de valeur 0 ou −1) peut simuler le type logique (qui n'existe pas en Basic 1000d). En effet le résultat d'une comparaison est un entier*1.

Exemples

La première exprc prend la valeur −1 (vraie) si $a \geq 125$, la deuxième aussi si $-7 \leq a < 12$.

```
a>=125
a IN [ -7,12[
a/2+12 NOT IN (b1, [1,145,2])
ch$="tre"
ch$ IN ("é",["a","z"],["A","Z"])
```

Les expressions (`c$` étant un v_ensemble et `a` une expression) :

```
a not in c$
not a in c$
not (a in c$)
```

sont identiques.

expra

```
[signe] terme { signe terme}
```

La définition expra représente une somme algébrique.

signe

```
    +
  | -
```

Exemple

Le signe + peut être omis.

```
+x-y-z
```

terme

```
  fact
| terme * fact
| terme / fact
| terme \ fact
| terme DIV fact
| terme MOD fact
```

La définition terme représente un produit de facteurs.

Exemple

Le terme suivant :

```
60/6*2
```

vaut 20 et est différent de :

```
60/(6*2)
```

qui vaut 5. Les opérateurs multiplicatifs \ et `div` représentent la division entière. Les expressions :

```
a DIV b
a \ b
divr(a,b)
```

sont égales. De même :

```
a MOD b
modr(a,b)
```

sont égales. Les opérateurs \, `div` et `mod` ne sont autorisés qu'entre `a` et `b` réels. Ils ne correspondent pas aux fonctions `div` et `mod`, mais seulement aux fonctions `divr` et `modr`.

fact

```
  primaire
| primaire ^[signe] primaire
```

La deuxième forme de fact, qui représente une exponentielle, peut être suivie du caractère de code ASCII 22, qui indique, en sortie, à l'imprimante la fin d'un exposant. Ce caractère est ignoré en entrée. Cela rend les fonctions `val` et `str$` inverses l'une de l'autre.

En exact l'exposant doit avoir une valeur entière dans $[-2^{15}, 2^{15}[$. En flottant l'exposant peut être non entier, et même complexe. Des expressions comme 100^(5/734) sont converties automatiquement en flottant. L'expression (-27)^(1/3) n'est acceptée que lorsque le littéral complexe est défini, bien qu'elle ait un sens en réel. Sa valeur n'est pas la racine cubique réelle de -27, mais la racine cubique complexe d'argument $\pi/3$.

Exemple

Dans le programme suivant, apparemment la même expression est calculée de façons différentes. L'expression `u` contenant un exposant flottant est calculée en 145 ms, par `exp(300~*log(1+i))`. Par contre, `w` est calculé plus

rapidement en 30 ms par multiplications, comme chaque fois que l'exposant est un entier*16.

```
complex i
w=(1~+i)^300
u=(1~+i)^300~
```

primaire

```
  Nombre exact ou flottant
| Index
| Variable de type var
| Littéral
| Nom inconnu;le type lit est attribué à nom
| Constante
| Variable d'état
| V_fonction
| label de V_fonction
| (expr)
| [expr]
```

Un index, une constante ou une variable est remplacé par sa valeur. Un littéral est gardé tel quel, sous forme symbolique. Une V_fonction ou un label de V_fonction, éventuellement suivi d'arguments, prend la valeur calculée par la fonction. (expr) ou [expr] (où les crochets n'indiquent pas un élément facultatif, mais sont réels) prend la valeur de expr.

Voyons maintenant quelques cas particuliers d'expr.

expr (sens restreint)

Dans les fonctions algébriques, comme `formf`(p), expr désigne seulement les expr exactes (flottants exclus).

nombre flottant

C'est une expr dont la valeur est un nombre flottant réel ou complexe.

poly

C'est une expr pouvant être écrite sous forme de polynôme. Par exemple (A^2-B^2)/(A-B).

nombre complexe exact

C'est une expr exacte qui ne contient pas d'autre littéral que le littéral complexe. Plusieurs formes peuvent représenter le même nombre, mais après assignation, il y a réduction à une forme canonique (voir `formc`).

Exemple

```
complex i
w=1/(1+i)
print w
print 1/(1+i)
```

Sortie (70 ms)

```
-1/2*i +1/2
  [i +1]^-1
```

nombre complexe

C'est un nombre complexe exact ou flottant.

réel exact

C'est un poly réduit à un nombre rationnel. Exemple 72/13

réel

C'est un nombre flottant réel ou un réel exact.

entier

C'est un réel exact à valeur entière.

entier*32 ou adresse

C'est un entier dans $[-2^{31}, 2^{31}[$.

adressepaire

C'est une adresse paire comme le nom l'indique.

entier*16

C'est un entier dans $[-2^{15}, 2^{15}[$.

ind

C'est un entier acceptable comme indice de tableau. Lors de la déclaration des tableaux, il ne peut excéder 2^{15} pour les littéraux ou variables, mais seule la place mémoire limite sa valeur pour un tableau d'index. Lors de l'utilisation des tableaux, il ne doit pas dépasser les dimensions déclarées.

littéral (généralisation du sens)

On utilisera le mot littéral dans le sens d'une expr dont la valeur est un seul littéral. Dans la suite, "littéral" sera toujours utilisé avec ce sens généralisé, et "type lit" qualifiera un littéral au sens strict.

Exemple

La variable `z`, de type var, est aussi un littéral (sens généralisé) puisque sa valeur est le littéral `x` (au sens strict, `x` étant de type lit).

```
z=x
print deg(a*x^7,z)
```

Sortie (15 ms)

```
7
```

Test des expressions mathématiques

COMPLEXP(p)

EXACTP(p)

FLOATP(p)

INTEGERP(p)

LITP(p)

MEMBERP(p, x)

POLYP(p)

RATNUMP(p)

Fonctions Test de *p*

p

expr

Ces fonctions demandent si l'expression *p* est d'une forme particulière et renvoient −1 si c'est vrai ou 0 (faux) sinon. Ainsi `floatp`(*p*) demande si *p* est flottant, `exactp`(*p*) si *p* est une expression exacte, `polyp`(*p*) si *p* est un polynôme, `complexp`(*p*) si *p* est un nombre complexe exact, `ratnump`(*p*) si *p* est un nombre rationnel exact, `integerp`(*p*) si *p* est un nombre entier exact, `litp`(*p*) si *p* est un littéral et `memberp`(*p*, *x*) si l'expression exacte *p* contient le littéral *x*.

Exemple

On dresse le tableau des valeurs renvoyées par ces fonctions pour diverses expressions. On a omis `complexp` qui est identique à `ratnump` pour un calcul en réels comme ici.

```
print tab(20);"floatp exactp polyp ratnump integerp lit
 p"
forv v in (pi,1/(1+x),x+1/17,x,19/3,2^58)
  print v;tab(21),floatp(v);"   ";exactp(v);"    ";poly
   p(v);
  print "    ";ratnump(v);"      ";integerp(v);"    ";l
   itp(v)
nextv
```

Sortie (635 ms)

```
                    floatp exactp polyp ratnump integerp litp
0.3141592654~ E+1    -1      0      0      0        0      0
 [x +1]^-1            0     -1      0      0        0      0
x +1/17               0     -1     -1      0        0      0
x                     0     -1     -1      0        0     -1
19/3                  0     -1     -1     -1        0      0
288230376151711744    0     -1     -1     -1       -1      0
```

Calcul des chaînes de caractères

Voyons maintenant les chaînes de caractères. Le mot “chaîne” désignera l’objet en mémoire, tandis que les mots “exprchaîne”, “elchaîne”, etc., définis ici, concernent leur écriture dans les programmes.

elchaîne

```
  " texte "
| variable de type char
| a;[a] L
| f;[a] S
| æ;[a] Z
| C_fonction
| label de C_fonction
| expr
```

La première forme permet de donner un texte explicitement. Les guillemets qui limitent le texte ne font pas partie de la chaîne. Les guillemets doublés `""` dans texte sont transformés en guillemet simple. Un nomi de type char (deuxième forme) prend la valeur de la variable. Les symboles a̲, *f* et æ prennent respectivement pour valeur les caractères de code ASCII 13, 32 et 0. A l’impression, ils correspondent à un retour chariot (13 et 0) ou un espace (32). Une C_fonction ou un label de C_fonction, éventuellement suivi d’arguments, prend la valeur calculée par la fonction. La forme expr, également admise, est convertie dans la chaîne de caractères `str$`(expr). Cette conversion automatique de expr n’est pas toujours possible avec un comparateur, `in` ou `not in` non parenthésé.

Exemple

On donne explicitement un texte contenant des guillemets.

```
c$="u$=""abcd"""
print c$
```

Sortie (15 ms)

```
u$="abcd"
```

Exemple

Dans les deux premières lignes, la conversion en chaîne est correcte. Dans la dernière forme, `"a"` a été décodé comme chaîne et affiché, puis le symbole < provoque une erreur dans la commande `print`.

```
c$=1<2
d$=("a"<"b")
print c$
print d$
```

```
print "a"<"b"
```

Sortie (80 ms)

```
 -1
 -1
a
*ERREUR* INSTRUCTION ILLEGALE
print "a"?
3.print "a"<"b"
```

exprchaîne

```
elchaîne { & elchaîne }
```

Le type exprchaîne est une chaîne utilisée dans les appels de fonction. Le symbole & réalise l'opération de concaténation. Le symbole de concaténation peut être omis devant *f* æ et a (Touches [a] S, [a] Z et [a] L).

Exemple

```
c$="espaces" f f & "puis" æ & "saut de ligne"
print c$
```

Sortie (35 ms)

```
espaces  puis
saut de ligne
```

virchaîne

```
[ exprchaîne ] {,exprchaîne}
```

Le type virchaîne intervient dans des commandes comme `music`. Les virgules sont alors équivalentes à &. Remarquer que la virchaîne vide est permise. Ainsi on peut écrire :

```
c$=
```

pour vider la variable c$.

fchaîne

Désigne une exprchaîne qui ne commence pas par une expr. Les noms de fichiers doivent être écrits comme fchaînes. Par exemple, les exprchaînes suivantes ne sont pas des fchaînes :

```
125
(1+x)^2
prog.prg
```

et les exemples suivants sont des fchaînes :

```
str$(125)
justc$((1+x)^2,50)
"prog.prg"
```

Les (v_ et c_) ensembles

Nous avons vu que les variables de type char peuvent avoir un contenu et une longueur arbitraires. Leur utilisation ne se limite pas à l'écriture de textes, et on peut envisager de s'en servir pour définir des structures.

Le Basic 1000d possède des fonctions internes permettant de traiter des structures de listes. Ces fonctions facilitent les extensions du Basic 1000d au calcul symbolique d'expressions irrationnelles ou transcendantes. Un exemple est fourni dans la bibliothèque MATH, permettant la manipulation d'expressions trigonométriques.

Dans cette section nous décrivons deux types de structures, représentant des ensembles.

Les v_ensembles contiennent des expr quelconque, des intervalles réels continus ou discrets. Les c_ensembles contiennent des chaînes quelconque et des intervalles lexicographiques.

On dispose de l'opérateur d'inclusion `in`, qui permet de déterminer si une expr ou exprchaîne appartient à un ensemble donné, de l'opérateur opposé `not in`, et des commandes de boucles `forv` ... `nextv` et `forc` ... `nextc` permettant d'effectuer une boucle sur les éléments d'un ensemble fini.

Les fonctions `vset$` et `cset$` permettent de créer des ensembles, et d'autres fonctions permettent de retrouver les éléments d'un ensemble, ou de modifier un ensemble.

Exemple

Le segment `[-1,1]` représente l'ensemble des réels $x : -1 \leq x \leq 1$. La progression `[-5,5,3]` est l'ensemble des nombres -5, -2, 1 et 4. Le v_ensemble `(0,1,[-5,5,3])` est l'ensemble des nombres 0, 1, -5, -2, 1 et 4. Noter que des éléments répétés sont possibles. Le c_ensemble `("un","deux,"trois")` est un ensemble de trois chaînes.

```
print 1/2 in [-1,1]
print 1 in (0,1,[-5,5,3])
print ("six" not in ("un","deux","trois"))
```

Sortie (60 ms)

```
-1
-1
-1
```

v_ensemble

```
  ( velsg {,velsg} )
| segment
| progression
```

```
  | exprchaîne
```
La forme exprchaîne doit avoir une structure de v_ensemble.

velsg

```
    expr
  | segment
  | progression
```

segment

Représente un intervalle continu de nombres réels. Il y a 4 formes possibles, qui font intervenir les symboles [et], comme dans la notation mathématique usuelle, et deux réels a et b.

```
    [a,b]
  | [a,b[
  | ]a,b]
  | ]a,b[
```
La forme du crochet indique si a et/ou b font partie de l'intervalle. Si $a > b$ l'intervalle est vide.

progression

La progression définie par les nombres réels a, b et c est l'ensemble fini des nombres réels a, $a + c$, $a + 2c$, $a + 3c$, ... qui ne dépassent pas la valeur b. C'est donc une progression arithmétique, mais le pas c peut être négatif. Cette progression se donne par :

```
[a,b,c]
```
où [et] désignent ici vraiment des symboles.

c_ensemble

```
    ( celsg {,celsg} )
  | csegment
  | exprchaîne
```
La forme exprchaîne doit avoir une structure de c_ensemble.

celsg

```
    exprchaîne
  | csegment
```
L'exprchaîne sera prise comme élément du c_ensemble.

csegment

Représente un intervalle de chaînes, suivant l'ordre lexicographique. Il y a 4 formes possibles, qui font intervenir les symboles [et], et deux exprchaînes a et b.

```
    [a,b]
  | [a,b[
  | ]a,b]
  | ]a,b[
```

La forme du crochet indique si a et/ou b font partie du csegment. Les csegments $[a, b]$ et $[b, a]$ sont identiques, à la différence des segments réels.

VSET$(velsg {, velsg})

CSET$(celsg {, celsg})

C_fonctions Construction d'un v_ ou c_ensemble

Les fonctions `vset$` et `cset$` permettent de construire des v_ et c_ensembles, par exemple pour conserver de tels ensembles dans des variables de type char. Les définitions données plus haut des v_ et c_ensembles ne sont possibles qu'après le mot clef `in`. Les chaînes représentant des ensembles peuvent être concaténées pour représenter la "réunion" des ensembles. Ainsi, les v_ensembles `c` et `cp` suivants sont identiques :

```
char c,cp
c=vset$(1)&vset$([3,5])
cp=vset$(1,[3,5])
```

Exemple

En mettant l'ensemble dans une variable, on évite d'avoir à le récrire à chaque test d'inclusion par `in`.

```
char c
c=cset$("bleu","rouge")
distingo
print ("bleu" in c)
print ("Bleu" in c)
nodistingo
print ("Bleu" in c)
print ("Bleu" in ["avant","centre"])
c=vset$(0,1,[20,50[)
print 7*pi in c and 0 in c and 50 not in c
print 3 in c&vset$([2,4,1/2])
```

Sortie (120 ms)

```
-1
 0
-1
-1
-1
-1
```

Les t_ensembles

Cette section qui décrit la structure des v_ et c_ensembles, ainsi que des fonctions permettant leur utilisation, peut être sautée en première lecture. Un t_ensemble est une chaîne formée par la concaténation de N ($N \geq 0$) t_éléments. Un t_élément est une chaîne possédant un type, qui est un entier dans $[0, 2^{16}[$. Le codage mémoire d'un t_élément est réalisé par :

longueur	contenu
2	le type
4	la longueur L de la chaîne
L	la chaîne
0 ou 1	un octet nul ou absent, sans signification qui permet d'atteindre une adresse paire

ESET$(t, x {, ti, xi})

C_fonction Construction d'un t_ensemble

t, ti

entier $\in [0, 2^{16}[$

x, xi

exprchaîne

La forme `eset$`(t,x) renvoie le t_élément construit avec l'exprchaîne `x` possédant le type `t`. Si plus de deux arguments sont donnés, `eset$(t, x, t1, x1, ...)` renvoie le t_ensemble `eset$(t, x) & eset$(t1, x1) & ...` Il est possible d'utiliser pour `x` (ou `x1`, ...) un autre t_ensemble, et de créer ainsi des structures contenant d'autres structures. Le type associé à chaque élément peut être utilisé pour indiquer si l'élément est un entier, un nom de fonction, un polynôme, etc.

Exemple

La C_fonction `eset_bis$`(t,x) renvoie le même t_élément que `eset$`(t,x). Dans le cas où la longueur de x est impaire le dernier octet superfétatoire, qui donne au t_élément une longueur paire, est `chr$`(0).

```
          t=random(2^16)
          c$=conc$(j=0,random(100) of chr$(random(2^8)))
          ift eset_bis$(t,c$)=eset$(t,c$) print "ok"
          stop
eset_bis$:function$(index t,char x)
          value=mkz$(t,2) & mkz$(len(x),4) & x
          ift len(x) mod 2 cadd value,æ
          return
```

Sortie (235 ms)

```
ok
```

Codage d'un v_ensemble

Chaque velsg d'un v_ensemble est représenté par un, deux ou trois t_éléments. La table suivante donne les types des t_éléments. Elle indique, par exemple, que le velsg $[a, b]$ est codé par la concaténation de 2 t_éléments :

```
eset$(2,mkx$(a)) & eset$($103,mkx$(b))
```

velsg		a	b	c
a		1		
$[a, b[$		2	3	
$]a, b[$		\$102	3	
$[a, b]$		2	\$103	
$]a, b]$		\$102	\$103	
$[a, b, c]$	si $c > 0$	4	5	6
$[a, b, c]$	si $c < 0$	\$104	\$105	6

Codage d'un c_ensemble

Chaque celsg d'un c_ensemble est représenté par un ou deux t_éléments. Voici les codages utilisés :

celsg	a	b
a	\$8001	
$[a, b[$	\$8002	\$8003
$]a, b[$	\$8102	\$8003
$[a, b]$	\$8002	\$8103
$]a, b]$	\$8102	\$8103

ELEMENTN(E)

V_fonction Nombre de t_éléments du t_ensemble E

ELEMENTY(E, k)

V_fonction Type du k-ième t_élément du t_ensemble E

ELEMENT$(E, k)

C_fonction Exprchaîne du k-ième t_élément du t_ensemble E

ELEMENTV(E, k)

V_fonction Expr du k-ième t_élément du t_ensemble E

CDR$(E)

C_fonction Renvoie le t_ensemble obtenu en ôtant le premier t_élément du t_ensemble E (E ne doit pas être vide)

E

exprchaîne ayant la structure d'un t_ensemble

k

entier $k > 1$

Les fonctions `elementn`, `elementy` et `element$` permettent de décomposer complètement un t_ ou c_ensemble.

Exemple

Lorsque les éléments d'un c_ensemble sont des textes imprimables, la procédure `printc` permet d'écrire sa décomposition.

```
        c$=cset$("seul",["un","six"])
        printc c$
        stop
printc:procedure(char E)
        local datai elementn(E) index i
        ift i=0 return
        for i=1,i
          print/h/ "(";elementy(E,i);",";element$(E,i);")"
        next i
        return
```

Sortie (100 ms)

```
(  8001,seul)
(  8002,un)
(  8103,six)
```

La fonction `elementv`(E,k) renvoie `cvx(element$`(E,k)) à condition que cela ait un sens. Avec `elementn` et `elementy` elle permet de décortiquer un v_ensemble.

Exemple

La procédure `printv` permet d'écrire la décomposition d'un v_ensemble.

```
        c$=vset$((a-1)^2,[1,10])
        printv c$
        stop
printv:procedure(char E)
        local datai elementn(E) index i
        ift i=0 return
        for i=1,i
          print/h/ using "($$####_,";elementy(E,i);
          print using "#)";elementv(E,i)
        next i
        return
```

Sortie (135 ms)

```
(  $1,a^2 -2*a +1)
(  $2,1)
```

```
($103,10)
```

Lorsque les t_ensembles sont utilisés comme listes, `cdr$` est analogue à la fonction CDR du langage Lisp. La fonction CAR s'obtient par `element$`(E,1).

Exemple

Pour sortir les éléments d'un v_ensemble sans progression on écrit le premier élément, on l'ôte par `cdr$` et on réitère jusqu'à ce que l'ensemble soit vide.

```
c$=vset$(1,2,3)
repeat
  print elementv(c$,1);
  c$=cdr$(c$)
until c$=""
```

Sortie (50 ms)

```
1  2  3
```

Exemple fichier disque

Les nombres 11 et 12 et la chaîne "abc" sont sauvegardés, sous la forme d'un t_ensemble, dans le fichier disque T.MUL par `save$`. Le t_ensemble est relu par `load$`, et ses éléments sont affichés.

```
save$ "t.mul",vset$(11,12) & cset$("abc")
char c
c=load$("t.mul")
print elementv(c,1)
print elementv(c,2)
print element$(c,3)
```

Sortie

```
  11
  12
abc
```

Assignation

nomi = ...

LET nomi = ...

Commande Met une valeur dans une variable ou index
Le mot clef `let` est en général omis.

Cas des variables de type var

Si nomi est de type var, la forme de l'assignation est :

```
nomi=expr
```

Nous avons déjà vu que si l'option `factor` (resp `develop`) est validée, il y a conversion en type factorisé (resp polynôme si c'est possible).

Exemple

La fonction `formf` renvoie $(X+1)^2$ mais dans l'assignation cette forme est développée. Par contre dans `print` le développement de `formf` n'est pas effectué.

```
develop
W=formf(X^2+2*X+1)
print W;formf(X^2+2*X+1)
```

Sortie (115 ms)

```
X^2 +2*X +1   [X +1]^2
```

Exemple

La dernière assignation est faite sous forme factorisée, mais sans rechercher les facteurs comme dans `formf`.

```
factor
W=formf(X^2+2*X+1)
PRINT W
W=X^2+2*X+1
PRINT W
```

Sortie (175 ms)

```
[X +1]^2
[X^2 +2*X +1]
```

Cas des index

Si nomi est de type index**size*, l'assignation se fait par :

```
nomi=entier
```

où l'entier signé doit pouvoir tenir dans *size* bits.

Cas des variables de type char

Si nomi est de type char, l'assignation se fait par :

```
nomi=virchaîne
```

Exemple

```
char CH
CH="(1+X)^2="æ,(1+X)^2
print CH
```

Sortie (35 ms)

```
(1+X)^2=
 X^2 +2*X +1
```

Cas nomi de type inconnu

Si nomi est un nom inconnu (et non indicé), Basic 1000d lui donne le type char ou index s'il se termine par un symbole déclaré par `implicit`. Sinon le nom prend le type var. Ensuite, l'assignation est effectuée selon le type du nom.

Exemple

Les symboles $ et % sont respectivement de types implicites char et index.

```
i%=2^31-1
c$="zyx"
v=2^31.5
print c$;i%;v
type c$,i%,v
```

Sortie (200 ms)

```
zyx  2147483647  0.3037000500~ E+10
c$ est de type char
i% est de type index
v est de type var
```

Assignation par Read et Data

READ nomi {, nomi}

DATA E1 {, Ei}

RESTORE [label]

Commandes Initialisation de variables et index

La commande `read` initialise les variables et index avec les expr ou exprchaînes `Ei` des lignes `data`. Le pointeur de lecture des `data` est mis sur le premier `data` qui suit `label` par `restore label` ou en début de source par défaut.

Exemple

```
    char Z
    read Z
    read W, I
    restore dat
    read J
    data un texte sans virgules , der((1+X+X^2)^3,X)
dat:'Ce label ne pointe pas forcément un data
    data 1000
```

```
print Z
print W
print I;J
```

Sortie (140 ms)

```
un texte sans virgules
  6*X^5 +15*X^4 +24*X^3 +21*X^2 +12*X +3
  1000  1000
```

Les assignations effectuées dans cet exemple équivalent à :

```
Z="un texte sans virgules"
W=der((1+X+X^2)^3,X)
I=1000
J=1000
```

Exemple

Comme les expr données dans `data` sont calculées, on peut initialiser la variable `a` avec un polynôme. Par contre, les exprchaînes ne sont pas calculées et l'initialisation par `data` diffère de l'assignation par =.

```
read a,c$
data x^2+min(1,2)
data left$("abc",2)
print a
print c$
c$=left$("abc",2)
print c$
```

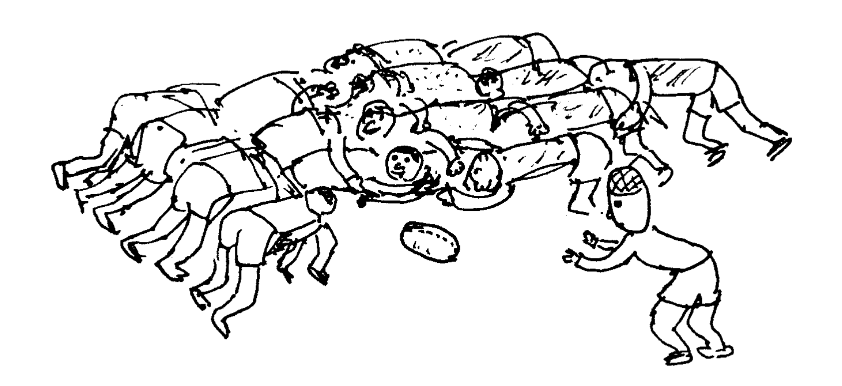
Sortie (555 ms)

```
  x^2 +1
left$("abc",2)
ab
```

Lorsque `read` et `data` sont utilisés dans un programme de la bibliothèque, l'utilisation de `restore` est obligatoire.

10

Entrée/Sortie



Sorties écran et imprimante

RESOLUTION r

Variable d'état Résolution

r

entier*16

La variable d'état `resolution` a quatre valeurs possibles 0, 1, 2 et 3. Pour chacune de ces valeurs r, la table suivante donne le nombre de lignes y et le nombre de caractères par ligne x (pour l'éditeur et la commande `print`), le nombre de couleurs c et les dimensions $x_{\max}$, $y_{\max}$, de l'écran en pixels.

r	y	x	c	$x_{\max}$	$y_{\max}$	
0	25	40	16	320	200	Basse résolution
1	25	80	4	640	200	Moyenne résolution
2	25	80	2	640	400	Haute résolution
3	50	80	2	640	400	Haute résolution

La commande

```
resolution r
```

met la résolution de même parité que r compatible avec le moniteur. Il est également possible de changer la résolution par [s] Tab dans la fenêtre Basic ou par `print chr$`(15), mais alors que la commande `resolution` vide l'écran, ces deux autres façons essaient de conserver les données écrites sur l'écran.

RESOLUTION0

Constante Résolution du bureau

La constante `resolution0` a la valeur de `resolution` au lancement du Basic 1000d. Dans le cas des moniteurs couleurs, la résolution opposée à `resolution0` présente des inconvénients graves lors de l'utilisation de la souris ou de l'AES. Pour vous éviter ces désagréments, Basic 1000d se remettra parfois en résolution `resolution0`.

Exemple

Un programme écrit pour la basse résolution peut commencer par les instructions de cet exemple. La résolution de lancement est remise, puis, si la résolution n'est pas conforme, le programme s'arrête après avertissement.

```
resolution resolution0
if resolution
  message "Basse Résolution|exigée"
```

```
    stop
  endif
  'suite du programme
```

Δ Le changement de résolution effectué par le Basic se situe au niveau du xbios. Comme la station de travail VDI a été ouverte avec la résolution de lancement, l'utilisation des couleurs avec l'autre résolution conduit à de nombreuses bizarreries. En voici un exemple, après lancement en basse résolution. Remplir une surface avec l'index de couleur 1 puis lire l'index de couleur d'un des points de la surface, avec `vpoint` (qui équivaut à un appel de la fonction 105 du VDI). En basse résolution, le résultat est 1, comme il se doit, mais en résolution moyenne on obtient 6 (qui n'est même pas un index valable).

LOCATE l, c

Commande Fixe la position du curseur sur l'écran

CURSC c

Variable d'état Colonne du curseur

CURSL l

Variable d'état Ligne du curseur

POS(virchaîne)

CSRLIN(virchaîne)

V_fonctions Position du curseur

c

entier $\in [0,79]$ (ou $\in [0,39]$ si `resolution`=0) (colonne)

l

entier $\in [0,24]$ (ou $\in [0,49]$ si `resolution`=3) (ligne)

Les valeurs $c = l = 0$ correspondent au coin supérieur gauche de l'écran.

Exemple

L'affichage a lieu sur la ligne 20, à partir de la colonne 10.

```
locate 20,10
print cursc;cursl
```

Sortie (25 ms)

```
   10  20
```

La commande `locate` de cet exemple équivaut à :

```
cursc 10
cursl 20
```

Les fonctions `pos` et `csrlin` peuvent s'écrire avec ou sans argument. L'argument est effectivement calculé s'il est donné. Le `print` de l'exemple ci-dessus équivaut à :

```
print pos(0);csrlin
```

ou

```
print pos;csrlin("x")
```

CURSH h

Variable d'état Haut de l'écran

h

entier $h \in [0, 16]$ (ou $h \in [0, 41]$ si `resolution`=3)

Après la commande

```
cursh h
```

les h premières lignes de l'écran sont fixes. Les lignes en dessous constituent la fenêtre de sortie et se déroulent au cours des impressions. Dans la fenêtre Basic ou après `clear`, la variable d'état `cursh` vaut 4, ce qui laisse 4 lignes fixes pour le damier.

Exemple

Le programme écrit les 10 lignes supérieures, avec `cursh` égal à 0. Ces lignes sont fixées lorsque `cursh` est égal à 10, et l'impression des 100 lignes suivantes fait défiler la fenêtre de sortie qui est formée des 15 lignes inférieures de l'écran.

```
cursh 0
print /c/conc$(i=0,9 of "ligne"&i&" fixe"a)
cursh 10
for i=1,100
  cursc random(20)
  print "Défile";i
next i
```

Sortie (4390 ms)

VIDEOINVERSE [k]

VIDEONORMAL [k]

Variables d'état

k

entier*16

Après l'une des 4 commandes (`g` entier pair) :

```
videoinverse
videoinverse g
videonormal g
print chr$($15)
```

on obtient l'impression en mode inverse. Après l'une des 4 commandes (`u` entier impair) :

```
videonormal
videoinverse u
videonormal u
print chr$($17)
```

on obtient le retour en impression normale. Les valeurs de ces variables d'état sont $70 et $71 en modes inverse et normal respectivement.

Exemple

Sur l'écran, la première ligne de la sortie est affichée en blanc sur noir.

```
videoinverse
print/h/videoinverse
videonormal
print/h/videonormal
```

Sortie (30 ms)

```
70
71
```

SHOWM [k]

SHOWC [k]

SHOWCM [k]

HIDEM [k]

HIDEC [k]

HIDECM [k]

Variables d'état Visibilité souris et curseur

k

entier*32

Les variables d'état `showm`, `showc`, `showcm`, `hidem`, `hidec` et `hidecm` renvoient toutes la même valeur entière x. Si x `and` $8000 vaut 0 la souris est visible, cachée sinon. Si x `and` $80 vaut 0 le curseur est visible, caché sinon. Pour faire apparaître/disparaître la souris, le curseur ou les deux, on peut utiliser la variable de nom convenable comme commande, sans préciser k. On peut aussi utiliser indifféremment une quelconque des variables avec la valeur de k correspondant à l'état souhaité. Seule est prise en compte la valeur de $x =$ `k and` $8080 suivant les règles ci-dessus.

Exemple

On cache la souris pendant 3 s, puis on la remet (visible ou cachée) comme en entrée.

```
push hidem
hidem
pause 3000
hidem pop
```

On notera que l'appel du clavier (`keyget`, `keytest`, `input`) ou le retour à la fenêtre Basic remet le curseur et la souris. D'autre part, dans `print`, la souris et le curseur sont localement cachés et rétablis à la fin du `print`.

Exemple

Comme la commande `print/h/ showm and $8080` écrit toujours 8080, il est nécessaire, pour écrire `showm`, de lire sa valeur en dehors du `print`.

```
hidecm
push showm
print /h/pop and $8080
ift keytest
push showm
print /h/pop and $8080;showm and $8080
```

Sortie (45 ms)

```
8080
0  8080
```

Δ Lors des calculs la fréquence du curseur peut ralentir. C'est normal, parce que le clignotement du curseur n'est pas géré par interruption VBL par le système, mais par le Basic 1000d, qui pendant les calculs s'occupe un peu moins du curseur. Cette complication a été rendue nécessaire par les insuffisances du GEM qui n'arrive pas à faire cohabiter un curseur clignotant et la souris (il laisse des bouts de souris ou de curseur lors des croisements).

PAGE texte

Commande Saut de page et titre imprimante

La commande `page` n'a d'effet que lors de la sortie de la source par les commandes du menu PRINTER. Elle force un saut de page, et écrit `texte` en haut de chaque nouvelle page.

PAGE_WIDTH k

Variable d'état Nombre de caractères par ligne imprimante

PAGE_LENGTH k

Variable d'état Nombre de lignes par page imprimante

LPOS

V_fonction Position de la tête imprimante

k

entier*16

Les variables d'état `page_width` et `page_length` servent pour l'impression de la source. De plus `page_width` est utilisée par la fonction `lpos`, et donc aussi lors de la tabulation dans `lprint`.

Le compteur servant à repérer la position de la tête imprimante, `lpos`, est géré comme suit. Il est remis à zéro après envoi des octets 0, 12 ou 13, ou lorsque sa valeur devient égale à `page_width`. L'octet 8 décrémente sa valeur, les autres caractères l'incrémentent. On notera que le compteur ne représente la position physique de la tête d'impression que si des séquences de commandes ne sont pas utilisées et que si `page_width` correspond à l'imprimante utilisée.

Exemple

La commande d'impression `lprint` n'envoie pas d'instruction de changement de ligne lorsque le compteur `lpos` atteint la valeur `page_width`. C'est

l'imprimante ou le programmeur qui doit s'en charger. Dans l'exemple, donner une largeur de page de 7 ne suffit pas pour imprimer des lignes de 7 caractères.

```
push page_width
page_width 7
for i=1,5
  lprint lpos;
next i
page_width pop
lprint æ;page_width;page_length
```

Sortie imprimante (90 ms)

```
0  3  6  2  5
80  59
```

HARDCOPY

Commande Copie d'écran

La commande `hardcopy` utilise la commande système, qui peut aussi s'appeler par appui sur [a] Help.

Exemple

L'appel `xbios($21)` de la première instruction sert à configurer le système pour une copie d'écran sur une imprimante Epson (960 points par ligne). Un cercle tracé sur l'écran est copié. Noter `cursh 0` pour vider tout l'écran, et `hidecm` pour cacher la souris et le curseur.

```
ift xbios($21,4)
cursh 0
hidecm
cls
circle 100,100,80
hardcopy
```

CLS

Commande Efface l'écran

La commande `cls` n'efface pas le menu. Seule la partie en dessous de la ligne `cursh` est effacée et le curseur est placé à gauche de la ligne `cursh`.

PRINT [/ {opt} /] liste

Commande Ecrit sur l'écran (périphérique "VBS:")

LPRINT [/ {opt} /] liste

Commande Sortie imprimante (périphérique "LBS:")

Dans la section sur la gestion des fichiers, nous verrons que les commandes `print` et `lprint` peuvent être déroutées (par exemple `print` peut envoyer des données vers un fichier disque, `lprint` vers "NUL:" pour supprimer l'impression). Aussi, pour que les explications qui suivent restent valables pour un périphérique quelconque, nous utilisons les notations "VBS:" et "LBS:", pour indiquer les périphériques de sortie écran et imprimante.

liste

```
{,[ el ]} [,]
```

Indique les données à sortir ou des spécifications de mise en page. Les virgules "," dans liste peuvent être remplacées par des points-virgules ";".

el

```
  expr
| exprchaîne
| TAB(n)
| USING u$
```

Ci-dessus les déterminants `using` et `tab` sont des mots clefs, `n` est un entier*16 > 0 et `u$` une exprchaîne.

opt

désigne une des 8 lettres suivantes (majuscule ou minuscule) E, I, C, A, D, H, B et O.

E

Met en exposant

I

Met en indices

Les options E et I ne sont actives que si le périphérique de sortie est "LBS:". Elles sont étudiées avec les sorties imprimantes un peu plus bas.

C

Vider l'écran, ou saut de page avant la sortie des données. L'option C n'est active que sur les périphériques "VBS:" et "LBS:". Elle provoque l'envoi de `chr$`(12) en début de commande.

A

Pause après la sortie de 21 − `cursh` (ou 46 − `cursh` si `resolution` = 3) lignes (moins un caractère). On peut alors continuer ou revenir à la fenêtre Basic de l'éditeur (en appuyant sur A). Cette option fonctionne mal si vous affichez des mouvements de curseur autre que saut de ligne. L'option A n'est active que sur le périphérique "VBS:".

Exemple

Il faut 14 écrans en résolution 3 pour écrire l'expression :

```
print/a/(1234+x)^100
```

Sortie (19 s pour la première page)

D

Avec l'option D, pendant la commande `print`, l'écriture et lecture des nombres est en base dix. Cependant la valeur de la variable d'état `base` n'est pas modifiée, et on peut donc l'écrire.

Exemple

```
base sixteen
```

```
        print /d/base;1000;$1000
        print base;1000;$1000
```

Sortie (50 ms)

```
16  1000  4096
10  1000  1000
```

Si `base` est modifiée pendant le décodage des arguments du `print`, l'option D est désactivée, comme dans l'exemple suivant où la C_fonction `hexa` change la base.

```
        print /d/ hexa;§100
        print /d/ base
        stop
   hexa:function$
        base $10
        return
```

Sortie (85 ms)

```
64
16
```

H

L'option H est l'analogue de l'option D pour la base seize.

Exemple

Vide l'écran et écrit §1000 en hexadécimal.

```
        print /CH/ §1000
```

Sortie (50 ms)

```
3E8
```

B

L'option B est l'analogue de l'option D pour la base deux.

Exemple

Le nombre 12.34 (décimal) est écrit en base 2. Cette écriture fait intervenir un nombre infini de chiffres après la virgule, mais on montre facilement qu'à partir d'un certain rang ces chiffres se répètent avec une période de 20 chiffres. On utilise `using` pour écrire le nombre dans un format qui met cette période en évidence. Noter que les 4 derniers chiffres, qui semblent infirmer la périodicité, sont en fait inexacts par suite de l'arrondissage.

```
        c$=string$(20,"#")&chr$(32,60)
        c$="####.#"&string$(4,c$)
        print/b/using c$,§12.34
```

Sortie (365 ms)

```
1100.010101110000101000111
      10101110000101000111
      10101110000101000111
      10101110000101001000
```

O

L'option O est l'analogue de l'option D pour la base huit.

Mode direct

Lorsque la commande `print` (périphérique "VBS:") est effectuée en mode direct, on vide d'abord la ligne texte où le résultat va être écrit, ce qui est utile quand on réutilise un `print` déjà sur l'écran.

Sortie par morceaux

Les expr et exprchaînes de `liste` sont sorties immédiatement après évaluation.

Exemple

Le premier `print` sort en un morceau, le deuxième en 3 morceaux.

```
print cursc&chr$(32,20)&cursc
print cursc;chr$(32,20);cursc
```

Sortie (75 ms)

```
0                                   0
0                                   23
```

Tabulation

Lorsqu'une virgule "," précède un élément `el` de `liste`, il y a d'abord sortie d'espaces pour que le compteur de tabulation devienne un multiple de 16. Il n'y a pas de tabulation lorsque un point-virgule remplace cette virgule. Le compteur de tabulation vaut `pos` ou `cursc` pour le périphérique "VBS:", `lpos` pour le périphérique "LBS:" et est un compteur interne local à la commande `print` ou `lprint` en cours pour les autres périphériques. Ce compteur est mis à zéro après envoi de `chr$`(0) ou de `chr$`(13). On notera que dans la sortie imprimante, il ne représente la position de la tête d'impression que si des séquences de commandes ne sont pas utilisées. Dans le cas des périphériques autres que "vbs:" et "lbs:", le compteur local est géré comme `lpos` (voir cette fonction), mis à part qu'au début du `print` ou `lprint` ce compteur est initialisé à zéro, et qu'il n'y a pas de longueur de ligne maximum.

Le déterminant `tab`(n) qui peut être suivi indifféremment de "," ou ";" force le compteur de tabulation sur la valeur n. Cela est réalisé par sortie d'espaces, éventuellement précédée d'un passage à la ligne si le compteur était supérieur à n. Seul `modr`(n,256) est pris en compte.

Sauf si `liste` se termine par "," ou ";", il y a passage à la ligne suivante après la commande. Les octets 13 puis 10 sont alors envoyés au terminal de sortie à la fin de la commande.

Exemple

La sortie est effectuée dans un fichier virtuel "mem:" identifié avec la variable c$. Le `tab` est équivalent à `tab`(44). La chaîne c$ est formée de 1000 espaces mis par `spc`, des octets 13 et 10 mis par `tab` (le compteur de tabulation

est remis à zéro), de 44 espaces également mis par `tab`, des 3 octets du mot end, et de deux octets 13 et 10 de fin de `print`.

```
open "O",1,"mem:",c$
print #1;spc(1000);tab(300);"end"
print len(c$)
```

Sortie (120 ms)

```
1051
```

Sortie sur l'écran (périphérique "vbs:")

Les caractères sont sortis sur la page de façon très semblable à ce que l'on obtient en tapant au clavier dans la fenêtre Basic. Ainsi lorsque l'impression arrive en bout de ligne écran, elle continue ensuite sur la ligne écran suivante et la ligne texte est rallongée. Par exemple, exécutez :

```
print/c/"A"&conc$(i=33,255 of chr$(i))
```

On obtient une ligne texte formée de plusieurs lignes écran. Cette ligne est maintenant modifiable dans la fenêtre Basic. Par exemple, déplacez le curseur sur la ligne et tapez sur [c] Delete. Vous avez ainsi fait passer la ligne dans le tampon d'effacement, comme vous pouvez le vérifier en tapant Undo.

Les caractères de codes 0-$1F ont des actions spécifiques au Basic 1000d pour la sortie sur l'écran ("VBS:"). Cependant, une grande compatibilité avec la sortie "CON:" a été conservée. L'appel de l'émulateur VT52 du système reste de toutes façons possible par le canal "CON:", mais compromet la réutilisation de l'écran.

Exemple

La troisième instruction modifie, selon le canal "CON:", la première ligne affichée. Avec 3 séquences Escape, elle déplace le curseur vers le haut, vers la droite, puis elle efface la fin de ligne. La commande `close` remet le canal "VBS:". Après exécution la ligne affichée ne peut pas être réutilisée. Déplacer le curseur sur "a" et taper une touche (en mode insertion) révèle les caractères écrits en "VBS:" par la première instruction.

```
print "abcdefghijklm"
print_dev "con:"
print chr$(27)&"A"&chr$(27)&"C"&chr$(27)&"K"
close
```

Sortie (40 ms)

```
a
```

Exemple

Le programme donne deux façons d'effectuer l'effacement de l'exemple précédent, tout en préservant la réutilisation de l'écran. Après affichage de la première ligne, le code 1 déplace le curseur vers le haut, le code 3 vers la droite et le code 8 efface la fin de ligne. La deuxième façon utilise simplement les séquences Escape du canal "VBS".

```
print "abcdefghijklm"
```

```
print chr$(1);chr$(3);chr$(8)
print "abcdefghijklm"
print chr$(27)&"A"&chr$(27)&"C"&chr$(27)&"K"
```

Sortie (70 ms)

```
a
a
```

Codes d'impression 0–$1F

La table des codes d'impression suivante s'obtient par la commande Printing du menu HELPS (F3). La commande du Basic `print chr$(n);`, pour $n < 32$, effectue le mouvement décrit ici si n est dans la table, et sinon, pour les codes $11, $12, $16, $18, $1A, et $1C à $1F, n'a aucun effet sur l'écran. La touche du clavier qui provoque le même mouvement dans la fenêtre Basic est aussi indiquée.

n	touche	
0	CR	Curseur au début de la ligne texte suivante
1	↑	Curseur vers le haut
2	↓	Curseur vers le bas
3	→	Curseur vers la droite
4	←	Curseur vers la gauche
5	Home	Curseur au début de la ligne `cursh`
6	Delete	Efface sous le curseur
7		Sonnette
8	[s] Delete	Efface la fin de la ligne texte
9	Tab	Curseur sur la position de tabulation suivante
A	[s] ←	Curseur en début de ligne texte
B	[s] ↓	Curseur en bas à gauche de l'écran
C	Clr	Efface l'écran en dessous de la ligne `cursh`
D	CR	Curseur au début de la ligne texte suivante
E	[s] →	Curseur en fin de ligne texte
F	[s] Tab	Change la résolution
10	Ins	Insère une ligne vide
13	Backspace	Efface devant le curseur
14	[s] Backspace	Efface le début de la ligne texte
15		Vidéo inverse
16		Fin d'exposant pour l'imprimante
17		Vidéo normale
19		Caractère suivant sans mouvement
1B		Début des séquences Escape (appel VT52)

Sur le canal "VBS:", la sortie d'un symbole Atari de code $k < 32$ est effectuée par l'envoi des deux octets $19, k. En fait, il suffit d'utiliser la fonction

`chrp$`(k) qui crée une chaîne formée de ces deux octets.

L'envoi du code §13 ou 0 provoque la terminaison d'une ligne texte, et l'affichage continue sur la ligne écran suivante.

Exemple

On écrit en vidéo inverse, puis le caractère Atari de code 1. On peut remplacer `chrp$`(1) par `chr$`($19) & `chr$`(1).

```
print chr$($15);"inverse";chr$($17);" normal ";chrp$(1)
```

Sortie (25 ms)

`inverse normal` ↑

Séquences Escape

Les séquences Escape sont analogues à celles de l'émulateur VT52 lorsque `cursh` est nul. Si `cursh` n'est pas nul, les lignes au dessus de `cursh` ne sont pas affectées. La table suivante donne les séquences qui peuvent suivre `chr$`($1B), et leurs effets, toujours accompagnés par les touches du clavier correspondantes. Le premier code suivant $1B est représenté par son symbole (ainsi A correspond à $41).

séquence	touche	
A	↑	Curseur vers le haut
B	↓	Curseur vers le bas
C	→	Curseur vers la droite
D	←	Curseur vers la gauche
E	Clr	Efface l'écran en dessous de la ligne `cursh`
H	Home	Curseur au début de la ligne `cursh`
I		Curseur vers le haut avec défilement
J		Efface tout l'écran après le curseur
K	[s] Delete	Efface la fin de ligne
L	Ins	Insère une ligne vide
M		Efface la ligne et remonte le bas de l'écran
Y y x		Positionne le curseur
b z		Couleur d'écriture
c z		Couleur du fond
j		Sauvegarde la position du curseur
k		Remet le curseur sur la position mémorisée
l		Efface la ligne
p		Vidéo inverse
q		Vidéo normale

Dans la séquence Y y x, les codes y et x sont les coordonnées du curseur augmentées de 32. Dans les séquences b z et c z, la donnée $z \in [0, 15]$ est le code de la couleur. Remarquer que les séquences d, e, f, o, v et w ne sont pas permises parce que incompatibles avec le traitement interne du curseur et de l'écran.

Sortie sur l'imprimante (périphérique "LBS:")

Vérification

Si l'imprimante n'est pas connectée, il y a immédiatement sortie d'un message d'erreur.

Longueur de ligne

La commande `lprint` n'envoie pas d'octets de changement de ligne, pour les lignes plus longues que `page_width`. Il faut donc régler l'imprimante pour qu'elle effectue elle-même le changement de ligne (c'est le réglage usuel). Cependant, il est utile de définir la variable d'état `page_width` avec la valeur de l'imprimante, pour un fonctionnement correct des tabulations.

Options E et I

Exemple

L'impression obtenue ressemble à la notation mathématique usuelle, avec exposants et indices.

```
lprint /EI/ "Z(1,7)=";X^3+X^2+1
```

Sortie imprimante (70 ms) :

$$Z_{1,7} = X^3 + X^2 + 1$$

Avec l'option E, la chaîne `C` envoyée à l'imprimante est modifiée avant envoi par :

```
C=change$(C,"^", chr$(27) &"S0", chr$(22), chr$(27) & "T")
```

c'est à dire que le signe "^" est converti en la séquence faisant passer l'imprimante en mode exposant et le caractère 22, qui indique la fin de l'exposant, en la séquence retour à l'impression normale. De même avec l'option I, ce sont les caractères "(" et ")" qui sont convertis en instructions de passage en indice et retour en impression normale. La chaîne `C` est modifiée par :

```
C=change$(C, "(", chr$(27)&"S1", ")", chr$(27) &"T")
```

Les conversions imprimantes utilisées conviennent aux imprimantes compatibles Epson. Elles peuvent être modifiées par poke, dans une table accessible par `systab`. On peut transmettre jusqu'à 8 octets.

adresses	valeurs pour imprimante Epson ou compatible	
`systab` + 46	"^"	caractère à changer
`systab` + 47	3, 27, "S0"	le premier octet est la longueur
`systab` + 56	22	
`systab` + 58	3, 27, "T"	
`systab` + 66	"("	

Codes envoyés à l'imprimante

D'autres octets sont modifiés avant l'envoi vers l'imprimante, même sans les options E et I. Voici la table des conversions :

origine	envoyé
0	$D, $A
$19, *k*	*k*
autre *n*	*n* inchangé

En pratique, pour envoyer l'octet *k* il suffit d'exécuter :

```
lprint chrp$(k);
```

où le ";" inhibe l'envoi du $D, $A terminal.

USING u$

Déterminant

u$

exprchaîne

Le déterminant `using` permet de formater la sortie des expr ou exprchaînes qui suivent `u$` dans `liste` du `print`.

Formatage des exprchaînes

La table suivante indique les symboles actifs pour formater les chaînes.

	rôle
&	fait tout sortir
!	fait sortir le premier caractère
\ \	fait sortir les premiers caractères
_	symbole suivant dans `u$` inactif

Exemple

Le format \23\ sort 4 caractères, en rajoutant des espaces si nécessaire. Seul le nombre de symboles entre les deux "\" est significatif, leur valeur ne jouant aucun rôle.

```
print using "_&: & ","abcdefg","xyz"
print using "_!: ! ","abcdefg","xyz"
print using "_\xx_\: \23\ ","abcdefg","xyz"
```

Sortie (80 ms)

```
&: abcdefg &: xyz
!: a !: x
\xx\: abcd \xx\: xyz
```

Formatage des expr

La table suivante indique les symboles actifs pour formater les expressions. x désigne un caractère quelconque.

	rôle
#	chiffre
*	chiffre
.	point décimal
^	exposant
~	tilde
	espace
,	virgule
+	signe (préfixe)
–	signe (suffixe)
$x	préfixe x
_x	désactive x

Exemple

Dans `"###.####"`, le "." fixe la position du point décimal écrit, et les quatre `#` après le "." fixent le nombre de chiffres décimaux écrits. Le même format a été appliqué aux deux nombres. Noter que les séparateurs "," et ";" devant ces nombres sont identiques après `using`. Autrement dit, la tabulation est alors supprimée. Cependant, un ";" en fin du `print` évite un retour chariot, comme sans `using`. Noter aussi que la valeur 0 est toujours sortie sous la forme "0" sans décimales ou exposant.

```
print using "###.####";1;10;
print using "###.##",0,10
```

Sortie (55 ms)

```
1.0000 10.0000  0    10.00
```

Exemple

Voici un exemple de format exponentiel. La présence de "^" impose le format exponentiel. On peut insérer des espaces ou des virgules dans le format. Elles seront également insérées dans les nombres sortis. Le "~" permet d'attribuer une place au tilde indiquant une valeur approchée. Le nombre de chiffres avant le point décimal est fixé par les variables d'états `formatl` et `formatm` comme lors d'impressions sans `using`.

```
print using "# # # # # . # #~^^^^",100/3
```

Sortie (35 ms)

```
0 . 3 3~ E+2
```

Exemple

La valeur des variables d'état `format` et `formatx` est modifiée localement dans le `print` avant la sortie de chaque nouvelle expression, suivant le nombre

de chiffres demandés après la virgule. Cette modification se prolonge dans les fonctions appelées par `print`. Elle reste définitive (jusqu'à `clear`) en cas de sortie erreur avant la fin du `print using`. Inversement, si l'expression à écrire modifie les variables d'état `format` ou `formatx`, comme la fonction `modif` dans l'exemple, la sortie est déréglée.

```
        print using "##.##",formatx,format;
        print using "###.###",modif,formatx
        print format;formatx
        stop
  modif:function
        formatx 17
        value=formatx
        return
```

Sortie (90 ms)

```
 3.00 3.00 17.0000000000000000  4.000
 -11  0
```

Exemple

Le "$" initial place le symbole "F" suivant devant le nombre. Les "*" jouent le même rôle que les "#", mis à part que des "*" sont sortis devant le nombre affiché. Le signe "–" placé en fin du format réserve une place pour le signe du nombre. Une autre option permise est de placer "+" en tête du format, ce qui sort le signe même s'il est positif.

```
        print using "$F*** *** *** ***.-",-10^6
```

Sortie (30 ms)

```
*****F1 000 000.-
```

Exemple

Le premier "_" indique que "#" ne spécifie pas le format, mais doit être sorti tel quel. La partie "a^b=" est également recopiée. Noter que "^" n'est pas un début de format. Le format réduit à "#" n'est pas suffisant pour contenir le nombre 70. Un nombre est cependant toujours sorti en entier, donc parfois avec débordement du format. Les symboles après le format affichent "b.".

```
        print using "_#a^b=#b_.",70
```

Sortie (20 ms)

```
#a^b=70b.
```

Exemple

Les expressions symboliques voient leurs nombres formatés. Le nombre en tête (s'il y en a) subit entièrement l'action du format, les autres seulement l'effet du changement de la valeur de `formatx`.

```
        print using "#.## ##",(1+2*x)^3
```

Sortie (70 ms)

```
8.00 00*x^3 +12.0000*x^2 +6.0000*x +1.0000
```

Exemple

L'exemple suivant examine des cas où plusieurs formats sont donnés dans la chaîne c$. D'abord il y a trois formats numériques (les symboles “&”, “\” et “!” sont inactifs après “_”). Si on essaie de faire sortir une chaîne avec ce format, elle n'est pas sortie, mais le texte de c$ est sorti. Lorsque la sortie d'un seul nombre est demandée, on obtient également la sortie des caractères non actifs de c$ situés avant le deuxième format de c$. Lorsque la sortie de plus de nombres que n'en comporte c$ est demandée, c$ est réutilisé. Les exemples avec `using d$` mélangent chaînes et expressions.

```
c$=" [_& ####_^2 _+ ####_^2 = ####_^2 _\_! ]"
print using c$,"A"
print using c$,777
print using c$,3,4,5,5,12,13
d$="    \2345\### mars"
print using d$,"dimanche",9
print using d$,"l",10;
print using d$,"m",11
```

Sortie (240 ms)

```
 [& ####^2 + ####^2 = ####^2 \! ]
 [&  777^2 +
 [&    3^2 +    4^2 =    5^2 \! ] [&    5^2 +   12^2 =   13^2 \! ]
     dimanc  9 mars
     l      10 mars     m      11 mars
```

SCREEN$

C_fonction Sauvegarde écran

RSCREEN virchaîne

Commande Chargement écran

La fonction `screen$` renvoie une chaîne de 37256 octets contenant des informations sur l'écran et les paramètres du VDI/AES, ainsi qu'un mot de vérification (checksum). La commande `rscreen` remet l'écran et les paramètres VDI/AES du moment de l'appel de `screen$`. Elle n'est exécutée que si la chaîne donnée en argument est de bonne longueur, et son checksum correct. En pratique `rscreen` n'acceptera donc que les chaînes créées par `screen$`.

Exemple

L'écran avec l'ellipse et le menu est sauvegardé dans la variable c$. Il est remis par `rscreen` après l'effacement de l'écran.

```
ellipse 200,250,200,100
c$=screen$
cursh 0
cls
ift keyget
rscreen c$
ift keyget
```

Damiers, Menus et Evénements

CHECKER virchaîne

Commande Ecriture d'un damier Basic 1000d

La commande `checker` permet d'écrire des damiers, en haut de page, à la façon des menus et sous-menus de l'éditeur. Cette écriture ne modifie pas la position du curseur. Pour protéger un damier de `h` lignes contre le défilement de l'écran, on utilisera :

```
cursh h
```

L'argument de la commande est une chaîne formée d'unités, dont le rôle est déterminé par le premier caractère *s* (toujours obtenu par Alt+lettre). Voici les 3 unités qui permettent de remplir une case du damier :

s	type	exemple	8	4
o̲	[a] M	`o̲MaJusCL`	MAJUSCL	MJCL
ÿ	[a] O	`ÿOrDiNaIRE`	Ordinair	Odni
ö	[a] T	`öTelQuel`	TelQuel	TelQ

et les unités supplémentaires :

s	type	effet	exemple
≪	[a] C	Centrage	≪`titre`
≫	[a] V	Change vidéo	≫
a̲	[a] L	Change de ligne	a̲

Une case occupe 8 ou 4 (si `resolution`=0) caractères, et `checker` ne va pas écrire plus de caractères pour chacune des cases. Les colonnes 8 et 4 indiquent ce qui est affiché suivant la résolution. L'unité [a] M est écrite en majuscules, [a] O en minuscules sauf la première lettre majuscule et [a] T est écrite sans conversion. Dans la résolution 0, pour [a] M et [a] O, les lettres minuscules ne sont pas sorties. Les espaces sont significatifs.

Avant l'écriture de la case du damier, la vidéo est changée (entre normale et inverse), de sorte qu'il y a création du damier. La commande s'occupe également du changement de ligne toutes les 10 cases. L'unité [a] C sort le texte qui suit centré sur une ligne. Le caractère [a] V change la vidéo et le caractère [a] L fait passer à la ligne suivante.

Exemple

Damier de 7 lignes et attente cliquage. Pour sortir la sixième ligne en vidéo inverse, on inverse la vidéo ([a] V), on sort une case vide ([a] O) et on va à la ligne ([a] L). Essayez ce que donne le simple changement de ligne [a] L.

```
push clock,date,cursh
noclock
nodate
cursh 7
checker "ºMaJusCLÿOrDiNaIREöTelQuelÿSTOP", conc$(i=5,50
  of "º F"&justl$(i)),"≫ÿª","≪Exemple 7 lignes"
cls
do
  i=keyget-$1FF
  ift i in [1,50] print "Vous avez cliqué F";justl$(i)
  ift i=4 exit
loop
cursh pop
cls
date pop
clock pop
```

Exemple de gestion des événements

Le programme est mis en attente d'événement par la commande `on menu`, placée dans une boucle :

```
Do
  On Menu
Loop
```

Avant cette boucle, la commande `menu Ms(0)` permet de définir des menus déroulants GEM, et des commandes de la forme :

```
on menu <type d'événement> <instruction>
```

indiquent au programme quels sont les événements à surveiller et les actions à entreprendre lorsque l'événement survient. Ainsi `On Menu Sp1` stipule que lorsqu'une entrée d'un menu déroulant sera sélectionnée il faudra appeler la procédure Sp1. Les autres événements surveillés par le programme sont ici l'appui d'une touche du clavier (`on menu key`), un délai de 1000 ms (`on menu timer`), l'entrée de la souris dans un rectangle (`on menu mouse`) et l'appui d'un bouton (`on menu button`).

Les informations sur l'événement s'obtiennent par la fonction `menu(n)`. Par exemple, dans la procédure `Spb`, la position de la souris au moment du cliquage est donnée par `menu`(10) et `menu`(11).

```
char Ms(50)
hidec
For I=0,50
  Read Ms(I)
```

```
    ift Ms(I)="*" exit
  Next I
  Data DESK,En route tout le monde,----------,1,2,3,4,5,6
   ,
  Data Transprt,Train,Bateau,Diligence,Chameau,
  Data EXECUTE,Départ,Fin,
  Data ,*
  CURSH 0
  cls
  CURSH 4
  Menu Ms(0)
  On Menu Sp1
  On Menu Key Sp2
  On Menu Timer 1000 Sp6
  On Menu Mouse 0,0,0,100,100,100 Sp3
  Sp3
  Sp4
  On Menu Button 2,1,1 Spb
  Do
    On Menu
  Loop
Sp1:menu off
  ift menu(0)<11 return
  ift menu(0)>14 goto Sp12
  for I=11,14
    menu I,3
  next I
  menu menu(0),2
  return
Sp12:ift Ms(menu(0))<>"Fin" return
  menu kill
  stop
Sp2:locate 10,35
  print chr$(menu(14) and $ff)
  return
Sp3:f_type 2+random(2)
  f_style 1+random(12)
  pbox 0,100,100,200
  return
Sp4:f_type 2+random(2)
  f_style 1+random(12)
  pbox 150,100,250,200
  On Menu Mouse 1,1,150,100,100,100 Gosub Sp5
  return
```

```
Sp5:On Menu Mouse 1,0,150,100,100,100 Gosub Sp4
    return
Sp6:locate 0,70
    print time$
    return
Spb:f_type 2+random(2)
    f_style 1+random(12)
    pcircle menu(10),menu(11),50
    return
```

MENU a$(k...)

Commande Construction de l'arbre du menu

a$(k...)

nomi de type char

La commande `menu` utilise les variables du tableau `a$` à partir de `a$`(k...). Le tableau peut avoir un ou plusieurs indices, mais ici nous supposerons qu'il a un seul indice et que l'on utilise la commande :

```
menu a$(0)
```

Le tableau `a$`(k) doit être rempli avant l'appel de la commande `menu`, par exemple comme suit :

k	exemple	
0	`desk`	Titre du premier menu déroulant
1	`info`	Ligne d'information
2	`--------`	Séparatrice
3	`1`	Premier accessoire
...		
8	`6`	Sixième accessoire
9		Vide, marque la fin du premier menu
10	`files`	Titre du deuxième menu
11	`open`	Premier élément du deuxième menu déroulant
...		
x		Vide, marque la fin du menu
...		
m		Vide, marque la fin du dernier menu
$m+1$		Vide, marque la fin des données

Les éléments `a$`(k) commençant par "-" sont déconnectés (en gris). Par exemple, si on initialise `a$`(3) à `a$`(8) avec "-", les accessoires ne peuvent pas être appelés.

La commande `menu` provoque seulement la construction de l'arbre du menu, si elle est possible. Le menu ne sera dessiné qu'après la commande `on menu`.

Entrée n du menu

Nous désignons ainsi la $(n + 1)$-ième chaîne utilisée par la commande `menu a$(k..)`. Seulement certaines entrées sont représentées sur l'écran. Dans l'exemple ci-dessus, l'entrée 3, "1", correspond effectivement à une entrée du menu, mais l'entrée 9 est une chaîne vide.

A chaque entrée n du menu qui est réellement affichée correspond un objet déterminé par son identificateur i dans l'arbre du menu. Savoir passer de n à i, ou inversement de i à n, est important dans la gestion des menus. Ces transformations sont assurées par les fonctions suivantes.

MENU_ID(i)

OB_ID(n)

V_fonctions Identificateurs d'objets du menu

n, i

entier*16

Les fonctions `menu_id` et `ob_id` effectuent la conversion entre numéro d'entrée n du menu et identificateur i de l'objet dans l'arbre du menu (voir ci-dessus). Pour une entrée réelle, les deux fonctions sont inverses :

```
n=menu_id(i)
i=ob_id(n)
```

Par contre, si n ou i ne correspond pas à une entrée réelle du menu, ces fonctions renvoient -1, ou produisent une erreur si n ou i dépasse la valeur la plus grande possible. Pour l'exemple donné dans la commande `menu` ci-dessus, `menu_id`(9) renvoie ainsi -1.

MENU KILL

Commande Fin d'attente d'événements

La commande `menu kill` supprime l'effet des commandes

```
menu a$(k..)
on menu ...
```

et supprime la barre des menus (appel de `aes(30)`). Par contre, il n'y a pas effacement des menus sur l'écran.

MENU OFF

Commande Affiche un titre de menu en normal

Après cliquage d'une entrée d'un menu, le titre du menu s'écrit en vidéo inverse. La commande `menu off` récrit ce titre en vidéo normale (appel de `aes(33)`).

MENU n, x

Commande Change l'état de l'entrée n du menu.

n

entier*16

x

entier dans [0,3]

x	effet
0	efface la validation
1	met la validation
2	désactive (écrit en gris)
3	active (écrit en normal)

La commande `menu` n, x utilise `aes(31)` ($x = 0, 1$) ou `aes(32)` ($x = 2, 3$). Elle modifie le titre correspondant à l'entrée n du menu suivant la valeur de x. Elle équivaut à :

```
aes 31+divr(x,2), ob_id(n), modr(x,2), menu(-1)
```

ON MENU

Commande Attend un événement

La commande `on menu` doit être précédée de commandes de la forme :

```
on menu <type d'événement> <instruction>
```

qui sont décrites plus bas. Ces commandes définissent l'événement à surveiller, et spécifient une instruction (au sens général du Basic 1000d). Si l'événement surveillé se produit lors de la commande d'attente :

```
on menu
```

il y a retour de la commande `on menu` et l'instruction correspondant à l'événement est exécutée.

La durée de l'attente d'événements dans la commande `on menu` dépend de la commande suivante :

ON MENU TIMER n instr_timer

Commande Durée d'attente

n

entier*32

instr_timer

instruction en Basic

La commande `on menu timer` spécifie que la commande :

```
on menu
```

doit attendre un événement pendant n millisecondes. Si un événement se produit avant ce délai, il y a sortie de la commande `on menu` avant la fin du délai. Sinon, au bout des n ms, on considère qu'il s'est produit un événement `timer` et l'instruction instr_timer est exécutée.

Lorsque cette commande est absente, tout ce passe comme si on avait donné un délai nul et une instr_timer vide :

```
on menu timer 0
```

ce qui fait que l'attente dans la commande `on menu` est alors très brève, même en absence d'événement.

Lors de l'attente d'événements on doit donc, soit placer `on menu` dans une boucle (voir l'exemple introductif), soit spécifier un grand délai. Les deux méthodes sont à peu près équivalentes, y compris pour l'interruption par Break.

Exemple

L'exemple suivant revient à attendre 5 secondes.

```
on menu timer 5000 print " mtimer=";mtimer
on menu
```

Sortie

```
mtimer=  5010
```

ON MENU instruc

Commande Surveillance des menus déroulants

instruc

instruction en Basic 1000d

La commande `on menu instruc` spécifie l'instruction qui sera effectuée si une entrée du menu est sélectionné.

Exemple

Affiche l'entrée sélectionnée dans un menu.

Il ne peut y avoir plus de trois accessoires, seul le deuxième est actif. Comme le menu est construit à partir de `C$`(1), l'entrée du menu sélectionnée est contenue dans `C$`(`menu`(0)+1). Les titres des menus (dans la barre) sont tronqués à 8 caractères, mais les autres entrées peuvent être plus longues. Les guillemets dans "ligne d'information" évitent que l'apostrophe soit prise comme marque de début de commentaire.

```
cursh 0
cls
hidec
dim C$(50)
for I=1,50
  read C$(I)
  ift C$(I)="?" exit
next I
data titre1,"ligne d'information",- - - - - - - - - - - -
  - -,-1,2,-3,
data huitcaractèresauplus,-choix1,CHOIX2,
data TITRE3,CHOIX3,-|-|-|-,CHOIX4,
data ,?
menu C$(1)
on menu exit
do
  on menu
```

```
loop
cursh 4
print/c/C$(menu(0)+1)
```

ON MENU KEY instr_key

Commande Surveillance du clavier

instr_key

instruction en Basic 1000d

La commande `on menu key` spécifie l'instruction qui sera effectuée si une touche du clavier est actionnée. La touche appuyée s'obtient par `menu`(14), et les touches spéciales par `menu`(13).

Exemple

Ecrit les touches tapées au clavier. Arrêt 30 s après la dernière touche appuyée.

```
on menu key print chrp$(modr(menu(14),2^8));
on menu timer 30000 stop
do
  on menu
loop
```

ON MENU BUTTON click, masque, be instr_button

Commande Surveillance des boutons

click, masque, be

entier*16

instr_button

instruction en Basic 1000d

La commande `on menu button` spécifie l'instruction qui sera effectuée après un événement bouton. Les arguments *click*, *masque* et *be* sont placés dans `gintin`(1) à `gintin`(3) avant l'appel implicite `aes(25)` de la commande `on menu`. En pratique, seulement les formes suivantes semblent utilisables :

```
on menu button x,a,a instr_button
```

où la valeur de x ne joue aucun rôle et a indique le ou les boutons à tester (bit_0 gauche, bit_1 droit). Le retour d'événement se produit lorsque le bouton (gauche si $a = 1$, droit si $a = 2$ ou les deux ensemble si $a = 3$) est appuyé.

Exemple

L'appui sur le bouton gauche dessine une boîte en mode graphique inverse (`pbox` après `graphmode` 3), une touche quelconque arrête le programme.

```
on menu button 0,1,1 gosub p
on menu key stop
do
  on menu
loop
```

```
p:graphmode 3
    origin menu(10), menu(11)
    pbox 0,0,60,40
    return
```

ON MENU MOUSE m, drap, x, y, dx, dy instr_mouse

Commande Surveillance de la souris

m

0 ou 1

Il est possible de définir deux événements souris indépendants par deux commandes différant par la valeur de m.

drap

0 ou 1

0 souris dans la boîte
1 souris hors de la boîte

x, y

entier*16

coordonnées absolues du coin supérieur gauche de la boîte

dx, dy

entier*16

largeur et profondeur de la boîte (en pixels)

instr_mouse

instruction en Basic 1000d

La commande `on menu mouse` spécifie l'instruction qui sera effectuée si la souris est dans la boîte (si $drap = 0$) ou au contraire en dehors de la boîte (si $drap = 1$).

Exemple

On surveille si la souris est à l'intérieur ou à l'extérieur de la boîte 100×100 dont le sommet supérieur gauche est le point $(0, 100)$. L'un des deux événements est toujours vrai, et un carré ou un disque est tracé suivant le cas. Une touche quelconque arrête le programme.

```
     graphmode 3
     hidec
     on menu mouse 0,0,0,100,100,100 dans
     on menu mouse 1,1,0,100,100,100 hors
     on menu key stop
     do
       on menu
       pause 500
     loop
dans:procedure
     pbox 0,100,100,200
```

```
        return
   hors:procedure
        pcircle 50,150,50
        return
```

ON MENU MESSAGE instr_message

Commande Surveillance des messages de l'AES

instr_message

instruction en Basic 1000d

La commande `on menu message` spécifie l'instruction qui sera effectuée si un message est généré par l'AES. Les messages concernent la gestion des fenêtre et les demandes de retracé de zone.

Exemple

Les appels AES concernent les fenêtres. `aes(100)` crée les fenêtres d'identificateurs `f` et `h`. Une fenêtre est ouverte par `aes(101)`, fermée par `aes(102)` et supprimée par `aes(103)`. Les coordonnées du rectangle intérieur de la fenêtre sont obtenues par `aes(104)`. Les variables d'état `f_type` et `f_style` fixent les attributs de remplissage pour le tracé d'une boîte par `pbox`. Cliquer sur la case de pleine fenêtre renvoie le message `menu`(1)=23. Cliquer sur la case de fermeture renvoie le message `menu`(1)=22. Le message `menu`(1)=20 correspond à une demande de retracé.

```
        hidec
        aes 100,0,0,0,640,400
        f=gintout(0)
        aes 101,f,0,0,640,400
        on menu message fen
        on menu
        aes 100,4+2,40,40,400,300
        h=peekw(gintout)
        aes 101,h,100,100,80,60
        do
          on menu
        loop
    fen:select case menu(1)
        case =22
          aes 102,h
          aes 103,h
          aes 102,f
          aes 103,f
          cls
          stop
        case in (20,23)
          if menu(4)=f
```

```
      f_type 2
      f_style 1
    else
      f_type 3
      f_style random(12)+1
    endif
    aes 104,menu(4),4
    x=gintout(1)
    y=gintout(2)
    pbox x,y,x+gintout(3),y+gintout(4)
  endselect
  return
```

MENU(n)

V_fonction Informations sur les événements

n

entier $n \in [-2, 15]$

Après la commande `on menu`, la fonction `menu(n)` renvoie les sorties de l'appel implicite `aes(25)` nécessaires pour gérer l'événement.

n	
-2	adresse du tampon de message
-1	adresse de l'arbre du menu
0	entrée du menu qui a été cliquée = `menu_id`(`menu`(5))
1 à 8	message de 16 octets
9 à 15	paramètres `gintout`(0) à `gintout`(6)

Le tampon de message, ou `menu`(1) à `menu`(8), contient les messages des événements fenêtre. Voir L Besle (1986) p196–8 pour leurs significations.

Détaillons les sorties `menu`(9) à `menu`(15).

menu(9)

Le type t de l'événement (`on menu` T) qui s'est produit :

t	T
1	`key`
2	`button`
4	`mouse` 0
8	`mouse` 1

Si plusieurs événements sont simultanés, on obtient la somme des valeurs correspondantes. Les deux événements souris (`mouse`) correspondent aux deux possibilités pour m (voir `on menu mouse`).

menu(10)

Coordonnée x de la souris

menu(11)

Coordonnée y de la souris

menu(12)

Boutons actionnés

	boutons
0	aucun
1	gauche
2	droit
3	gauche et droit

menu(13)

Touches spéciales du clavier

	touches
1	Shift droite
2	Shift gauche
4	Control
8	Alternate

Si plusieurs touches sont appuyées, on obtient la somme des valeurs correspondantes.

menu(14)

Touche clavier appuyée

octet faible	`modr`(`menu`(14), 2^8)	code ASCII
octet fort	`divr`(`menu`(14), 2^8)	code Clavier

menu(15)

Nombre de cliquages

ALERT i, cm, b, cc, rep

MESSAGE cm

Commandes Boîte d'alerte

SURE?(cm)

V_fonction Demande confirmation

i

entier $i \in [0, 3]$

i	symbole d'alerte
0	aucun
1	!
2	?
3	stop

cm

exprchaîne

Message où les lignes sont séparées par "|". Au plus, `cm` peut comporter 5 lignes de 30 caractères.

b

entier $b \in [0, 3]$

b	bouton de sortie par Return
0	aucun
1	premier
2	deuxième
3	troisième

cc

exprchaîne

Texte des boutons radio, séparés par "|". Le nombre de boutons est limité à 3, et le texte de chaque bouton ne doit pas dépasser 8 caractères.

rep

nomi de type var ou index, contient le numéro du bouton cliqué.

La commande `alert` trace une boîte d'alerte et attend une réponse.

Exemple

Le programme affiche une forme d'alerte, avec symbole et bouton de sortie aléatoires, tant que le bouton Stop n'a pas été choisi.

```
do
```

```
      i=random(4)
      j=random(3)
      alert i," Exemple avec le |"&justc$("symbole"&i,17)&
       "|  et le bouton"&j,j,"1|2|Stop",rep
      print "rep=";rep
      ift rep=3 exit
    loop
```

La commande `message` est une forme simplifiée de la commande `alert`, sans symbole, et avec un seul bouton “OK”.

Exemple

```
    message "Exemple|de message"
```

La fonction `sure?` écrit le message `cm` dans une boîte d’alerte, avec symbole “!”, et deux boutons, OUI et ANNULER. Elle renvoie -1 (vrai) si le bouton OUI est sélectionné, et 0 (faux) si le bouton ANNULER est choisi.

Exemple

Noter comment l’utilisation de guillemets répétés `""` permet la création d’une chaîne contenant des guillemets.

```
    print sure?("Exemple : appel |Sure?(""Exemple ..."")")
```

Entrée clavier

INPUT [“texte” [,]] N1 {, Ni}

Commande Entrée d’expr et exprchaînes au clavier

LINE INPUT [“texte” [,]] NC1 {, NCi}

Commande Entrée d’exprchaînes par ligne au clavier

(Séparateurs)

Les “,” peuvent être remplacées par “;”, et la virgule après “texte” peut être omise, sans modification de l’effet des commandes.

N1, Ni

nomi de type var, char ou index

NC1, NCi

nomi de type char

Les commandes `input` et `line input` affichent d’abord “texte”, qui ne doit pas dépasser 98 caractères, (ou “input>” par défaut), puis attendent une entrée clavier suivie de [CR]. Dans la section sur la gestion des fichiers, nous verrons que le périphérique d’entrée peut être redéfini (par exemple, au lieu d’attendre

une entrée clavier, les données peuvent être lues sur l'entrée série ou sur un fichier disque). Dans le texte entré, les 3 caractères ,() et les guillemets "..." ont un rôle particulier, comme illustré dans l'exemple suivant. Si l'entrée est :

```
4,min(a,5),tac,"8,7(",chr$(65,9)
```

la commande

```
INPUT N1,N2,C1$,C2$,C3$
```

réalise les assignations :

```
N1=4
N2=min(a,5)
C1$="tac"
C2$="8,7("
C3$="chr$(65,9)"
```

L'expr `min(a,5)` est calculée, tandis que l'exprchaîne `chr$(65,9)` n'est pas calculée. La virgule et la parenthèse entre guillemets dans `"8,7("` ne sont pas actives. On notera que l'entrée des exprchaînes qui commencent par un guillemet supprime ce guillemet et s'arrête au premier guillemet suivant rencontré. Ainsi la commande `input` ignore les caractères `ign` (un message est écrit) dans l'entrée

```
"8,7("ign
```

Si on entre trop d'expressions, Basic 1000d indique que les entrées en trop sont ignorées. Si on n'en rentre pas assez, la commande réclamera de nouvelles entrées. Les initialisations par `read` et `data` ou par `input` suivent les mêmes règles. L'exemple précédent a un effet identique à :

```
read N1,N2,C1$,C2$,C3$
data 4,min(a,5),tac,"8,7(",chr$(65,9)
```

La commande `line input C$,D$` assigne la totalité de la ligne texte validée à la variable `C$`, même s'il y a des virgules. Elle redonne ensuite la main au clavier pour attendre l'entrée de la valeur de `D$`.

KEYGET

V_fonction Attend touche ou bouton

KEYTEST

V_fonction Teste le clavier ou les boutons

La différence entre ces fonctions est que `keyget`, mais pas `keytest`, attend l'appui sur une touche ou le cliquage d'un bouton. Toutefois, si un bouton est appuyé, `keytest` attend qu'il soit relâché pour revenir.

Codes renvoyés par KEYTEST et KEYGET

Ils sont aussi tabulés dans KEYBRD (F2) du menu HELPS.

0

Renvoyé par `keytest` si aucune touche ou bouton n'est appuyé.

1 à $FF

Ce sont les codes renvoyés par les touches du clavier qui ont des codes ASCII, ou encore par [a] nnn (ou [a] Hnn en hexadécimal).

$100 à $1FF

Les combinaisons de touches Control, Shift et Alternate avec des touches spéciales renvoient $100+v où v est donné en hexadécimal dans la table suivante. Les touches [ca] Ins et [ca] Home renvoient chacune deux codes en succession. Les combinaisons Control, Alternate et une lettre renvoient les codes $180 (si la lettre est A) à $199 (pour la lettre Z).

	↑	↓	→	←	Home	Ins	Del	Bs	Tab	Help	Undo	CR	Esc
	1	2	3	4	5	10	7	13	9	18	12	D	20
s	6	B	E	A	C	1A	8	14	F	11	1B	19	21
a							27	33	29		32	2D	22
sa							28	34	2F		3B	39	23
c	41	42	55	56	57	50	47	53	49	58	52	4D	24
cs	46	4B	5C	5D	5E	5A	48	54	4F	51	5B	59	25
ca	61	62	75	76	*xx*	*xx*	67	73	69	78	72	6D	26

$200 à $24F

L'appui sur la touche de fonction $k \in [1, 10]$ renvoie le code :

$$2^9 + 10(4c + 2a + s) + k - 1$$

où s, a, c valent 0 ou 1 suivant l'appui ou non sur Shift, Alt et Control. Le bouton gauche, cliqué au dessus de la ligne `cursh` renvoie les mêmes codes, ce qui permet de traiter facilement les attentes d'événement damier.

$280

Ce code est renvoyé par [c] +

$10000 à $24F31

Cliquer un bouton ($b = 1$ pour le gauche ou $b = 2$ pour le droit) en un point de la colonne $x \in [0, 79]$ (ou $x \in [0, 39]$ si `resolution`=0) et de la ligne $y \in [h, 24]$ (ou $y \in [h, 49]$ si `resolution`=3) où $h =$ `cursh`, renvoie le code :

$$2^{16}b + 2^8x + y$$

$50001 à $5FF1A

L'appui sur [c] $\{$lettre$\}$ renvoie le code :

$$5 \times 2^{16} + 2^8 s + p$$

où $p \in [1, 26]$ dépend de la première lettre (1 pour A, 26 pour Z) et où $s \in [0, 255]$ est la somme modulo 256 de $d(lettre)$ pour les lettres suivantes, avec $d(A) = 1$, $d(B) = 2$, ..., $d(Z) = 26$.

Exemple

Le programme écrit le code renvoyé par `keyget` en hexadécimal, et le caractère ASCII correspondant s'il existe. Lorsque la case Stop du damier est sélectionnée, `keyget` renvoie $200.

```
checker "ÿSTOP",string$(19,"ÿ"),"a≪Exemple KEYGET"
do
  x=keyget
  print/h/x;
  ift x in [1,2^8[ print using "(&)",chrp$(x);
  ift x=$200 stop
loop
```

INKEY$

C_Fonction Lecture du clavier sans attendre

La fonction `inkey$` teste le clavier, sans attente comme `keytest`, et renvoie une chaîne de 0, 1 ou 2 caractères. Si la touche appuyée a un code ASCII, la fonction renvoie le caractère correspondant. Si la touche appuyée n'a pas de code ASCII, la fonction renvoie la chaîne de deux caractères `chr$`(0) & `chr$`(k), où k est le code clavier de la touche. Si aucune touche n'est appuyée, ou si seulement les touches Control, Shift, Alternate ou Caps Lock sont appuyées, elle renvoie la chaîne vide.

Exemple

Le programme suivant permet d'examiner la chaîne renvoyée par `inkey$`. Arrêt par appui sur “S” ou “s”.

```
DO
  c$=INKEY$
  IFT LEN(c$)=1 print "ASC=";ASC(c$);" TOUCHE=";chrp$(
   asc(c$))
  IFT LEN(c$)=2 print "CLAVIER=";ASC(RIGHT$(c$,1))
  IFT c$ in ("S","s") stop
LOOP
```

Sortie (Touche A)

```
ASC=  65 TOUCHE=A
```

Sortie de sons

MUSIC virchaîne

Commande Musique, bruits, sons

La commande `music` permet de programmer facilement la fonction `xbios($20)`. L'argument virchaîne doit être composé de lettres éventuellement suivies d'un nombre k, et des parenthèses “(” et “)”. Chaque lettre, “(” et “)” produit une commande élémentaire paramétrée par le nombre k. Si le nombre k est omis, sa valeur est prise égale à 0.

Exemples

`A`	émet une note
`w100`	attend 101 × 20 ms
`v10`	met le volume sur 10 pour les canaux ouverts

Dans la commande `music` la base est localement dix, ainsi :

```
base §16
music "o20A"
```

met octave=§20 et joue la note §20. On peut cependant donner k en hexadécimal, par exemple la commande `music` ci-dessus équivaut à :

```
music "o$14 A"
```

Exemple

Avant de donner la description complète des sous-commandes, voici deux exemples de `music`. Nous décomposons la virchaîne pour la commenter.

```
music "o35 t5 M4J4F4 ACEFC2FA5 w14 H4M4J4F4C  EFH2JH5
 w14"
```

`o35` fixe la valeur attribuée à la note représentée par “A”.
`t5` la durée des notes est un multiple de $5 \times 20 = 100$ ms.
`A` joue la note 35 pendant 100 ms. Chaque lettre A–Z produit l'émission d'une note, pendant une durée dépendant du nombre suivant la lettre.
`M4` joue la note 47 pendant 500 ms.
`w14` pause de 300 ms.

```
music "(Aq1u2Dv8m11p500s10y9r2"
```

`(A` met la période de la note 36 (“A”). Après “(”, “A” n'émet pas une note, mais fixe la période.
`q1` met l'enveloppe 1.
`u2` ferme le canal A (les commandes précédentes concernaient le canal A) et ouvre le canal B.
`D` fixe la période (celle de la note “D”) du canal B.
`v8` met le volume du canal B à 8.
`m11` valide le canal A + bruit et le canal B.
`p500` fixe la période de l'enveloppe.
`s10` fixe la forme de l'enveloppe.
`y9` fixe le pas y de la boucle qui suit.
`r2` effectue une boucle qui envoie la suite des valeurs 1, $1+y$, $1+2y$, ..., 251 calculées modulo 256 dans le registre 2, ce qui fait varier la fréquence du canal B.

Les sous-commandes de MUSIC

Les sous-commandes de `music` écrivent dans les registres 0 à 13 du générateur de son YM2149, par l'intermédiaire de la fonction `xbios` $20. Cela est indiqué par $< r,\ k>$ (envoie k dans le registre r). D'autre part <$FF, k>,

<\$80, k> et <\$81, ...> indiquent des commandes de `xbios` \$20. Les canaux A, B, C sont désignés par l'entier $i \in [0,2]$.

Variables internes

La commande `music` utilise des variables internes, dont voici la liste avec les valeurs initiales au début de chaque commande `music`.

par=")**"**

Prend les valeurs "(" et ")" ce qui indique si on est en intérieur "(" ou extérieur ")" de parenthèses. Les commandes A–Z diffèrent suivant la valeur de cette variable.

note

Prend les valeurs entières dans $[0,95]$. La période correspondant à *note* est, en unités de 8×10^{-6} seconde, $p(note)$ qui est la partie entière de $3822 \times 2^{-note/12}$. Par exemple pour $note = 45$, $p(note) = 284$ correspond à une période de 2.272 ms et à une fréquence de 440 Hz environ.

octave=36

Prend les valeurs entières dans [0,95]. C'est le numéro de la note A.

tmps=8

Temps en unité 20 ms (au début 160 ms).

x=1, y=10, z=251

Limites et pas d'une boucle `for I=x,z,y` modulo 256.

unit=1

$unit = a_0 + 2a_1 + 4a_2$ où $a_i = \mathrm{bit}_i$ de *unit* vaut 0 (fermé) ou 1 (ouvert) et indique l'état du canal i. Initialement seul le canal A est ouvert.

env

Le bit_i de *env* est à 1, si l'enveloppe est validée pour le canal i.

Au début de `music` le volume est mis à 10 par <7, \$FE>, <8, 10>. En fin de `music`, les canaux sont fermés par <7, \$FF>, <\$FF, 0>.

Sous-commandes

(

Met par="("

)

Met par=")"

A k

...

Z k

Posons $n = 0$ pour "A", $n = 1$ pour "B", ..., $n = 25$ pour "Z" Cette commande définit $note = n + octave$ qui doit être dans $[0,95]$. La commande

dépose la période $p(note)$ dans les registres des canaux i ouverts (tels que bit_i de $unit = 1$). <$2i$, p_b>, <$2i + 1$, p_h> où p_h et p_b sont les octets haut et bas de $p(note)$.

De plus si par=")", les canaux ouverts sont validés, on attend $t = (k + 1)tmps$ (en unités de 20 ms), puis on ferme les canaux. Cela correspond à l'émission d'une note. <7, `not` *unit*> <$FF, *t*> <7, $FF>.

m k

$k \in [0, 63]$
<7, `not` *k*>. Pour $i \in [0, 2]$ le bit_i de k correspond à la validation du son sur le canal i, et le bit_{i+3} de k correspond à la validation du bruit (1=allumé 0=éteint). Par exemple `m9` valide le canal A et mixe le générateur de bruit.

n k

$k \in [0, 31]$
<6, *k*>. k est la période du bruit.

o k

$k \in [0, 95]$
Met $octave = k$ qui est le numéro de la note "A".

p k

$k \in [1, 2^{16}[$
<11, `modr`(*k*, 256)> <12, `divr`(*k*, 256)>. C'est la période de l'enveloppe.

q k

$k \in [0, 7]$
Valide l'enveloppe et met $env = k$. Pour les canaux i ouverts <$i + 8$, $10>.

r k

$k \in [0, 13]$, $k \neq 7$
Effectue la boucle de la fonction `xbios` $20 : <$80, *x*> <$81, *k*, *y*, *z*>, c'est à dire que dans le registre k on place, à raison d'une valeur toutes les 20 ms, la suite de valeurs x, $x+y$, $x+2y$, ..., z (modulo 256). Cette suite peut être infinie, si z n'est pas dans la suite. Il est surtout intéressant d'utiliser cette commande avec les registres 2, 4 et 6 ce qui donne des variations de fréquence.

s k

$k \in [0, 15]$
<13, *k*>. Met la forme du son (shape).

t k

Met $tmps = k$ (en unités de 20 ms).

u k

$k \in [0, 7]$
Met $unit = k$ qui définit les canaux ouverts et fermés.

v k

$k \in [0, 15]$

Pour les canaux i tels que le bit_i de *unit* soit 1 et le bit_i de *env* soit 0, met le volume : $<8+i,\ k>$.

w k

$k \in [0, 254]$

Attend $k+1$ (en unités de 20 ms). $<\$FF,\ k+1>$.

x k

$k \in [0, 255]$

Met $x = k$.

y k

$k \in [1, 255]$

Met $y = k$.

z k

$k \in [0, 255]$

Met $z = k$.

Exemples de boucles

```
music "(u3Am$13v15x71y255z35r0w6x91z55r2"
music "(o50Av15m9x50y1z200r0"
```

Souris

La visibilité de la souris, conditionnée par des variables d'état comme `showm` et `hidem`, a été étudiée en même temps que la visibilité du curseur.

MOUSE x, y, k

Commande Lecture de la position et des boutons de la souris

x, y, k

nomi de type var ou index

La commande `mouse` met dans x et y des entiers donnant la position de la souris ($x \in [0, 639]$ et $y \in [0, 399]$ en monochrome). Elle met dans k un entier qui

indique l'état des boutons. Le bit_i de k correspond au $(i+1)$-ième bouton.

k	bouton appuyé
0	aucun
1	gauche
2	droit
3	gauche et droit

Exemple

L'état de la souris est écrit. Dans la commande `print`, `chr$`(5) positionne le curseur au coin haut gauche de la fenêtre et `chr$`(8) efface la fin de ligne. Le clignotement de la souris est provoqué par `print`, qui cache la souris avant d'écrire les résultats. Arrêt par appui sur le bouton droit.

```
hidec
do
  mouse x,y,z
  print chr$(5);x;y;z;chr$(8)
  ift z=2 stop
loop
```

Exemple

On trace un cercle centré sur la souris. Arrêt en cliquant les deux boutons.

```
hidec
K=0
Do
  ift K<>1 Graphmode 3
  ift K=1 Graphmode 1
  ift K=2 Cls
  ift K=3 stop
  Mouse X,Y,K
  Circle X,Y,40
  Circle X,Y,40
Loop
```

SETMOUSE x, y

Commande Déplace la souris en x, y

x, y

entier*16

La commande `mouse` du Basic 1000d utilise implicitement `aes(79)`, et non `vdi(124)` (qui fonctionne imparfaitement avec l'AES). Cependant, la fonction 79 de l'AES n'est pas non plus exempte de défaut. Ainsi après la commande

`setmouse` elle renvoie des valeurs fausses tant que la souris n'a pas été déplacée. Le programme suivant explique cette difficulté.

```
          print/C/"La souris va être déplacée par SETMOUSE (appuy
           er sur une touche)"
          ift keyget
          print "Position initiale";MOUSEX;MOUSEY
          SETMOUSE MOUSEX+25,MOUSEY+25
          print "La nouvelle position est donnée exactement par V
           DI 124."
          print "Par contre, MOUSEX, MOUSEY ou AES 79 donnent la
           position avant déplacement :"
          L
          print "Maintenant, déplacer la souris, puis appuyer sur
            une touche."
          IFT KEYGET
          print "Les diverses indications coincident :"
          L
          stop
        L:VDI 124
          print "VDI 124 -->";ptsout(0);ptsout(1)
          print "MOUSEX/Y-->";MOUSEX;MOUSEY
          AES 79
          print "AES 79  -->";gintout(1);gintout(2)
          return
```

Sortie

```
La souris va être déplacée par SETMOUSE (appuyer sur une touche)
Position initiale  391  307
La nouvelle position est donnée exactement par VDI 124.
Par contre, MOUSEX, MOUSEY ou AES 79 donnent la position avant déplace
 ment :
VDI 124 -->  416  332
MOUSEX/Y-->  391  307
AES 79  -->  391  307
Maintenant, déplacer la souris, puis appuyer sur une touche.
Les diverses indications coincident :
VDI 124 -->  215  278
MOUSEX/Y-->  215  278
AES 79  -->  215  278
```

MOUSEX

MOUSEY

Variables d'état Position de la souris

Les commandes :

```
          mousex x
```

```
mousey y
```
déplacent la souris, comme
```
setmouse x,y
```
En tant que fonctions, `mousex` et `mousey` renvoient la position de la souris.

MOUSEK

V_fonction Etat des boutons

Le “k” de `mousek` est d’origine germanique (Knopf). La valeur renvoyée est identique à celle obtenue par la commande `mouse`.

Exemple

Le programme suivant est une amélioration de l’exemple de `mouse`. L’état de la souris n’est écrit que s’il est modifié, ce qui supprime le clignotement.
```
hidec
index x,y,k
do
  if mousex<>x or mousey<>y or mousek<>k
    mouse x,y,k
    print chr$(5);x;y;k;chr$(8)
    ift mousek=2 stop
  endif
loop
```
Dans le programme ci-dessus on peut remplacer :
```
mouse x,y,k
```
par :
```
x=mousex
y=mousey
k=mousek
```

DEFMOUSE t

DEFMOUSE ch

Variable d’état Forme de la souris

t

entier*16

t	Forme
0	flèche
1	barre
2	abeille
3	index
4	main

ch

exprchaîne de 74 octets contenant une forme personnelle

La chaîne `ch` est utilisée pour initialiser les mots $k = 0$ à 36 du tableau `intin` de l'appel système `vdi(111)`. On notera d'ailleurs qu'il est parfois plus simple d'utiliser directement la commande `vdi` 111 que `defmouse ch`.

Le masque et le motif dessinent un carré de 16×16 pixels. x et y sont les coordonnées du point d'action, relativement à l'origine du motif.

k	
0	x
1	y
2	1 ou -1
3	couleur du masque
4	couleur du motif
5 à 20	masque
21 à 36	motif

En tant que fonction, `defmouse` renvoie la valeur de t, ou 255 (forme utilisateur), correspondant à son dernier appel.

Exemple

Une forme utilisateur est définie, puis on fait défiler les 8 formes prédéfinies de la souris.

```
C$=mkz$(1,6)&mkl$(1)&mkz$(-1,32)&conc$(I=0,15 of mki$(
 -2^I))
defmouse C$
forv i in ([7,0,-1],0)
  print/c/"defmouse=";defmouse
  pause 1500
  defmouse i
nextv
```

Lecteurs de disques, répertoires et fichiers

On se reportera aussi au chapitre 6 où on trouvera quelques explications sur l'organisation des disques, ainsi que des éclaircissements sur la façon de changer de lecteur (exemple du disque virtuel).

CHDRIVE lec

Variable d'état Lecteur de disques par défaut

lec

Indique le lecteur. C'est soit un entier $\in [1, 16]$, soit une exprchaîne commençant par une lettre de A à P.
En tant que fonction, `chdrive` renvoie un entier de 1 à 16 (1 pour le lecteur A).

Exemple

On écrira, pour passer au lecteur B, une quelconque des commandes suivantes. Noter que la dernière forme ne change pas le répertoire.

```
chdrive 2
chdrive "B"
chdrive "b:\AA\"
```

DFREE(lec)

V_fonction Nombre d'octets libres sur le disque lec

lec

Indique le lecteur. C'est soit un entier $\in [0, 16]$, soit une exprchaîne commençant par une lettre de A à P. A la différence de ci-dessus dans `chdrive`, $lec = 0$ est possible pour désigner le lecteur courant.
Pour le lecteur courant, on peut écrire `dfree` au lieu de `dfree(0)`. Le nombre d'octets libres des disques A et B s'obtient aussi par la commande Dir du menu FILES.

Exemple

```
print dfree;" octets sur le disque ";chr$(chdrive+64)
```

Sortie (2170 ms)

```
159744 octets sur le disque A
```

DIR$(lec)

C_fonction Nom du répertoire du lecteur lec

lec

Comme ci-dessus dans `dfree`
La forme `dir$` équivaut à `dir$`(0).

Exemple

La sortie a été obtenue après `chdir "auto"`.

```
print dir$(0)
```

Sortie (20 ms)

```
\AUTO
```

MKDIR ch

RMDIR ch

CHDIR ch

Commandes Créer, effacer ou fixer le répertoire

ch

fchaîne

La commande `mkdir` (respectivement `rmdir`, `chdir`) crée (MaKe) (respectivement efface (ReMove), définit (CHange)) le répertoire (DIRectory) de nom contenu dans l'exprchaîne `ch`. La commande `rmdir` exige que le répertoire soit vide.

Exemple

La commande `mkdir` crée le répertoire SEST, ce qui est vérifié en affichant le contenu du disque par `files$`. Le répertoire par défaut est modifié par `chdir` et affiché par `dir$`. Enfin, le répertoire est effacé par `rmdir`.

```
mkdir "\SEST"
print files$
chdir "\SEST"
print "dir$=";dir$
chdir "\"
rmdir "\Sest"
```

Sortie (6055 ms)

```
[ A: ] *.*    720896 octets libres
     0+SEST         ,  3207 X.Z
dir$=\SEST
```

FSEL$(c1, c2 [, c3])

C_fonction Sélection de fichier

c1, c2, c3

fchaîne

La fonction `fsel$` utilise le sélecteur de fichier de l'AES (`aes(90)` ou `aes(91)` si on donne le titre `c3` et TOS $\geq$ 1.4). `c1` est un chemin de répertoire. Si `c1` est vide, cela correspond au répertoire par défaut. Si `c1` se termine par "\" comme par exemple `c1="A:\"`, on rajoute au nom de répertoire `*.*`. Ce rajout n'est pas effectué si par exemple `c1="A:\numer\*.z"`. La racine du disque s'obtient avec `c1="\"`. Le nom du fichier placé sur la sélection, `c2`, peut être vide.

La fonction `fsel$` renvoie une chaîne vide si la sélection est annulée, et le nom complet avec chemin (eg `"A:\numer\stui.z"`) sinon.

Exemple

On écrit le nom du fichier sélectionné.

```
print fsel$("","")
```

Sortie

```
A:\X.Z
```

FILES$(c1 [, k])

FILES$

C_fonction Liste un répertoire

c1

fchaîne

k

entier*32, par défaut $k = 0$

Le chemin `c1`, analogue à celui de `fsel$`, définit un ensemble de fichiers, par le jeu des jokers `*` et `?`. La fonction `files$` renvoie la liste de ces fichiers, accompagnés de diverses indications suivant la valeur de k. Cette liste est précédée du chemin par défaut, entre crochets, du chemin `c1` et de la taille disponible sur la disquette. Cette taille n'est sortie que si le disque est A ou B, ou si le bit_4 de k est mis.

Si $k = 0$, les noms de fichiers, précédés de leurs longueurs, sont groupés à raison de 4 par lignes. La commande :

```
print/A/FILES$
```

a le même effet que la commande `dir` du menu FILES de l'éditeur ou du Basic.

Si $k \neq 0$, les noms de fichiers sont disposés à raison d'une ligne par fichier, et suivis des éléments suivants (si le bit correspondant de k est 1).

bit	
0	attribut
1	longueur
2	date de création
3	heure de création
4	longueur libre sur disque

Les attributs d'un fichier sont codés dans un octet :

bit	
0	lecture seule
1	caché
2	système
3	racine
4	répertoire
5	refermé

Exemple

La commande imprime les noms, longueurs, dates et heures de création des fichiers d'extension Z du disque A.

```
lprint files$("A:\*.Z",%1110)
```

Sortie imprimante (2120 ms)

```
[ A: ] A:\*.Z    721920 octets libres
X.Z             3207 04/24/1986 03:51:08
```

Exemple

La C_fonction `fulldir`(p$) suivante analyse `files$`(p$,3) pour déterminer quels sont les sous-répertoires de p$. Puis la fonction `fulldir`, s'appelle ensuite elle-même pour analyser ces sous-répertoires. Noter le procédé utilisé pour ouvrir un nouveau canal *n* tout en assurant la réentrance de `fulldir`. On recherche simplement un canal non encore ouvert, (i.e. tel que `devty`(*n*)=0). Noter également l'utilisation des fichiers virtuels "MEM:", de `eof`, `line input` et `input$`.

Dans l'exemple, `fulldir`("A:\") renvoie la liste complète des répertoires et fichiers du disque A.

```
        c$=fulldir("A:\")
        print /a/c$
        stop
fulldir:function$(char p$)
        local index n char d$,f$
        value=files$(p$,3)
        for n=0,100
          ift devty(n)=0 exit
        next n
        open "i",n,"mem:",value
        line input #n,c$
        while not eof(n)
          c$=input$(12,n)
          f$=input$(3,n)
          ift left$(c$)<>"." and f$=" 10" cadd d$,fulldir(p$&
           justl$(c$)&"\")
          line input #n,c$
        wend
        close n
        cadd value,chr$(13)&chr$(10)&d$
        return
```

Sortie

```
A:\*.*          0 octets libres
DEGELITE.100 08      0
AUTO         10      0
BLOCKS       10      0
...
TRYSD16.FNT  00   4442

A:\AUTO\*.*          0 octets libres
.            10      0
```

```
 ..            10      0
 GDOS.PRG      00   8064
 ...
```

DIR [c1]

Commande Liste le répertoire c1

c1

fchaîne (défaut vide)

La commande `dir c1` est idendique à :

```
print/a/files$(c1)
stop
```

Elle liste des fichiers, comme la commande Dir de l'éditeur, et provoque un arrêt du programme.

Exemple

```
dir "a:\"
```

Sortie (2 s)

```
 [ A: ] a:\*.*    721920 octets libres
   3207 X.Z
```

EXIST(f)

V_fonction Teste l'existence du fichier f

f

fchaîne contenant le nom d'un fichier

Si le fichier existe la fonction `exist` renvoie −1, sinon 0.

Exemple

La fonction lirenom sélectionne le nom d'un fichier par `fsel$`, vérifie qu'il existe et renvoie ce nom. Dans la commande `checker` les caractères exotiques ≫, ≪ et a̲ sont obtenus par [a] V, [a] C et [a] L.

```
          print lirenom
          stop
 lirenom:function$
          checker "≫≪Quel est le fichier à charger?a̲a̲a̲≪"
          cls
          do
            value=fsel$("","")
            ift value<>"" ift exist(value) return
          loop
```

KILL f

Commande Efface le fichier f

f

fchaîne contenant le nom du fichier à détruire

La commande `kill` permet d'effacer un fichier disque. On peut aussi utiliser la commande Kill (menu FILES) de l'éditeur.

Exemple

Efface le fichier "x.z", puis sort en erreur parce que le fichier "y.z" n'existait pas.

```
kill "x.z"
kill "y.z"
```

Sortie (2 s)

```
*ERREUR* GEMDOS #-33
kill "y.z"?
2.kill "y.z"
```

NAME c1 AS c2

Commande Renomme un fichier

c1

fchaîne, nom de l'ancien fichier

c2

fchaîne, nouveau nom du fichier

La commande `name` change le nom d'un fichier. Le séparateur `as` entre c1 et c2 peut être remplacé par ",". L'identificateur de disque doit être le même pour les deux fichiers.

Exemple

Avant de sauvegarder c$ par `save$` dans le fichier "x.a", on modifie le nom de l'ancien fichier, s'il existe, en "x.bak". Noter que `kill` efface d'abord "x.bak", sinon `name` sortirait en erreur.

```
if exist("x.a")
  ift exist("x.bak") kill "x.bak"
  name "x.a" as "x.bak"
endif
save$ "x.a",c$
```

Gestion par fichier complet

VERIFY

V_fonction Vérifie lecture/écriture disque

La fonction `verify` relit le dernier fichier disque qui vient d'être lu ou écrit et compare son contenu avec la zone mémoire lue ou écrite. La fonction renvoie le nombre d'octets qui diffèrent, mais si la vérification est impossible, elle renvoie la valeur −1. Cette fonction est analogue à la commande de même nom du menu FILES, et peut aussi être utilisée après une opération disque de l'éditeur.

Exemple

La zone mémoire de 256 octets correspondant au contenu de c$ est sauvegardée, puis vérifiée. Après avoir modifié deux octets dans cette zone, la fonction `verify` prend la valeur 2.

```
c$=space$(256)
bsave "test",ptr(c$),256
print verify
pokeb ptr(c$)+20,1,2
print verify
```

Sortie (5835 ms)

```
0
2
```

LOAD [c]

MERGE [c]

SAVE [c]

Commandes Concernent la source

c

fchaîne

Les commandes `load`, `merge` et `save` effectuent les commandes homonymes de l'éditeur. Si le nom du fichier `c` (extension Z par défaut) est omis, elles demandent un nom. La commande `load` demande confirmation, ainsi que `save` dans le cas de recouvrement. Après ces commandes il y a retour à l'éditeur. On dispose aussi d'une commande `run c`, décrite ailleurs, qui charge et exécute un programme.

SAVE$ f, S

Commande Sauvegarde S sous le nom f

LOAD$(f, l)

C_fonction Contenu du fichier f

S

virchaîne

f

fchaîne représente le nom d'un fichier (extension DAT par défaut)

l

entier*32, nombre d'octets chargés

Après `C$=load$(f)` il y a chargement du fichier f en entier dans la variable C$. Si la mémoire disponible est insuffisante, seul le début du fichier f est chargé. Si *l* est donné, `load$` charge les *l* premiers octets du fichier f. Les deux instructions

```
save$ f,C$
C$=load$(f)
```

sont inverses l'une de l'autre. Si on utilise :

```
save$ f, S1
save$ f, S2
```

avec le même nom de fichier, le fichier créé avec S1 est recouvert par la deuxième instruction et on ne peut plus relire S1. Si on veut sauvegarder S1 et S2 il faut soit créer deux fichiers :

```
save$ f1, S1
save$ f2, S2
```

soit créer un fichier contenant les deux chaînes :

```
save$ f,S1,S2
```

Voir aussi l'exemple dans la section t_ensemble. Avec `load$` et `save$`, il n'y a pas besoin des commandes `open` et `close`. Cependant les fichiers étendus (comme "LST:", voir `open`) ne sont pas acceptés.

Exemple

La procédure `fcopy u,v` copie le fichier `u` sous le nom `v`.

```
     fcopy "a:x.a", "b:y.a"
     stop
fcopy:save$ @2,load$(@1)
     return
```

BLOAD f [, a]

Commande Charge le fichier f à l'adresse *a*

BSAVE f, a, l

Commande Sauvegarde le fichier f, avec les *l* octets en *a*

f

fchaîne, contient le nom du fichier

a

adresse, par défaut la dernière valeur utilisée par `bsave`

l

longueur en octets

Les cases “Load img” et “Save img” du menu FILES permettent d’exécuter les commandes `bload` et `bsave` à partir de l’éditeur.

Exemple

Sauvegarde puis charge l’écran. L’écran occupe 32000 octets à partir de l’adresse `peekls`($436) (ou `xbios(2)`).

```
print chr$($42,500)
bsave "T.IMG",peekls($436),32000
print /c/
bload "T.IMG",peekls($436)
i=keyget
cls
```

Gestion séquentielle ou sélective

Avant toute opération, il est nécessaire d’ouvrir un canal par `open`. En cas d’écriture sur disques, `close` permet de terminer la modification physique sur les disquettes.

OPEN mode, [#]n, nomdev [, rlen]

Commande Ouvre le canal *n*, correspondant au fichier ou périphérique *nomdev*, en accès *mode*

mode

exprchaîne

La table suivante donne les valeurs possibles de *mode* qui spécifie le mode d’accès du canal.

mode	`devty`	Mnémonique	
`"I"`	-1	Input	Lecture
`"O"`	-2	Output	Ecriture, nouveau ou efface l’ancien
`"A"`	-3	Append	Lecture et écriture, rajoute des données
`"U"`	-3	Update	Lecture et écriture
`"R"`	$k > 0$	Random	Accès sélectif

n

entier $\in [0, 100]$

Le symbole `#` devant le numéro du canal n est facultatif.

nomdev

```
    "MEM:",ch
  | fchaîne
```

Dans la première forme, `ch` est un nomi de type char qui sert de fichier virtuel "MEM:". Dans le cas du mode `"O"` :

```
open "O",1,"mem:",ch
```

la variable `ch` est d'abord vidée. Dans la deuxième forme, *nomdev* peut être un nom de fichier disque, ou le nom d'un périphérique pris dans la table suivante (sauf "MEM:"). On écrira par exemple "Vbs:", sans distinction entre majuscules et minuscules.

nomdev	`devid`	périphérique
MEM:	$k < -7$	Fichier virtuel (MEMoire)
NUL:	-3	Sortie NULle
LBS:	-2	Imprimante avec conversions (List BaSic)
VBS:	-1	Ecran ou clavier (Vidéo BaSic)
LST:	0	Imprimante, sortie brute sans conversions (LiST)
AUX:	1	RS232 (AUXiliaire)
CON:	2	CONsole
MID:	3	Musical Instrument Digital interface
IKB:	4	Intelligent KeyBoard
VID:	5	VIDéo
	$k > 5$	Fichier disque

rlen

entier*32 > 0

L'argument *rlen* spécifie la longueur d'un enregistrement dans le cas où le mode est "R". Il doit être omis dans les autres modes.

DEVTY([#]n)

V_fonction Mode d'accès du canal n

DEVID([#]n)

V_fonction Identificateur du canal n

n

entier $\in [0, 103]$

La fonction `devty` renvoie 0 si le canal est fermé. Si le canal est ouvert, la valeur renvoyée est indiquée dans la table des modes de la commande `open`. La fonction `devid` renvoie l'identificateur du périphérique, suivant la table donnée dans la commande `open`.

Le numéro du canal peut être non seulement un des canaux 0 à 100 définissables par `open`, mais aussi un des canaux 101 à 103 qui sont les canaux

utilisés par défaut par les commandes de la table suivante. Le mode d'accès des canaux 101 à 103 est fixe, mais le périphérique *nomdev* peut être redéfini par les commandes `input_dev`, `print_dev` et `lprint_dev`.

n	dans	`devty`	*nomdev*	redéfini par
101	`input` et `input$`	-1	VBS:	`input_dev`
102	`print` et `write`	-2	VBS:	`print_dev`
103	`lprint`	-2	LBS:	`lprint_dev`

INPUT_DEV nomdev

Commande Fichier implicite d'entrée (`input`)

PRINT_DEV nomdev

Commande Fichier implicite d'écriture (`print`)

LPRINT_DEV nomdev

Commande Fichier implicite d'impression (`lprint`)

nomdev

A la même syntaxe que dans `open`

Exemple

La sortie écran est déroutée vers "NUL:", ce qui supprime l'affichage par `print`. La sortie imprimante est déroutée vers l'écran ("VBS:" qui correspond à `devid` $= -1$). La commande `close` rétablit les sorties usuelles.

```
print_dev "nul:"
print x,y,z
lprint_dev "vbs:"
lprint using "devty=#  devid=#",devty(103),devid(103)
close
```

Sortie (60 ms)

```
devty=-2  devid=-1
```

Lorsque la sortie du `print` est déroutée vers un fichier disque, la commande `print_dev` joue le rôle de la commande `open`. Mais il faudra terminer la sortie par une commande :

```
close #102
```

ou :

```
print_dev "VBS:"
```

qui ferme le fichier.

CLOSE [[#]n]

Commande Ferme le canal n (tous les canaux par défaut)

n

Entier $n \in [0, 103]$

Noter qu'un espace entre `close` et `#` est nécessaire, car :

```
close#1
```

est, dans sa totalité, un nom valable du Basic. La fermeture des canaux 101 à 103 remet les valeurs naturelles des périphériques de ces canaux.

LOF([#]n)

V_fonction Longueur du canal

EOF([#]n)

V_fonction Test de fin de fichier

LOC([#]n)

V_fonction Pointeur du canal

SEEK [#]n [, i]

RSEEK [#]n [, i]

RELSEEK [#]n, i

Commandes Positionnement du pointeur de canal

n

entier $n \in [0, 103]$

Le canal n doit posséder un pointeur (disques ou “mem:”).

i

entier*32 (défaut $i = 0$)

La fonction `lof` renvoie la longueur du fichier utilisé comme canal n. La fonction `eof` renvoie -1 (vrai) si le pointeur du canal n est en fin, par exemple si le fichier a été entièrement lu, 0 (faux) sinon. La fonction `loc` renvoie le pointeur du canal n, qui est un entier ∈ [0 , `lof`(n)].

La commande `seek` place le pointeur du canal n en i. Si $i < 0$ la commande `seek` est identique à :

```
rseek #n,-i
```

La commande `rseek` place le pointeur du canal n i octets avant la fin du fichier. La commande `relseek` déplace le pointeur du canal n de i octets.

Exemple

Le pointeur du canal est placé en 0 après `open "i"` ou `seek` 1. Il est placé en fin de fichier après `open "a"` ou `rseek` 1.

```
print "lof  loc  eof"
c$="123456789"
open "i",#1,"mem:",c$
pointeur
relseek 1,5
pointeur
seek 1
pointeur
```

```
          rseek 1
          pointeur
          close 1
          open "a",#1,"mem:",c$
          pointeur
          stop
 pointeur:print using "##   ",lof(1),loc(1),eof(1)
          return
```

Sortie (195 ms)

```
  lof  loc  eof
   9    0    0
   9    5    0
   9    0    0
   9    9   -1
   9    9   -1
```

BPUT [#]n, a [, l]

Commande Ecriture sur un canal

BGET [#]n, a [, l]

Commande Lecture d'un canal

n

entier dans [0,103]

a

entier*32

l

entier*32 > 0 (défaut $l = 1$)

La commande `bput` écrit sur le canal n les l octets se trouvant à partir de l'adresse mémoire a. La commande `bget` écrit dans la mémoire, à partir de l'adresse a, l octets lus sur le canal n.

Exemple

Le haut de l'écran est sauvegardé dans le fichier virtuel "mem:" c$, puis ressorti sur l'écran en 7 tranches. L'adresse de l'écran est obtenue par `xbios(2)`.

```
open "O",#1,"MEM:",c$
cursh 0
cls
print conc$(i=32,255 of chr$(i))
bput #1,xbios(2),5120
close #1
cls
open "I",#1,"mem:",c$
for i=0,6
  bget #1,xbios(2)+i*5120,640
```

```
next i
close #1
i=keyget
```

INP(#n)

INP(k)

V_fonction Lecture d'un octet

OUT #n {, b}

OUT k {, b}

Commande Ecriture d'un ou plusieurs octets

n

entier $n \in [0, 103]$ numéro de canal

k

entier $k \in [0, 5]$ numéro du périphérique

b

entier*32 seul modr(b, 2^8) est envoyé

La fonction `inp` lit un octet sur le canal n ou sur le périphérique k. La commande `out` envoie un ou plusieurs octets vers le canal n ou le périphérique k. Les numéros k des périphériques sont les valeurs de `devid` données dans la table des périphériques de la commande `open` (voir aussi `bios(1)`). Les formes `inp`(k) et `out` k, ... ne nécessitent pas l'ouverture d'un canal par `open`. Par contre, veiller à ne pas omettre le signe `#` dans les formes `inp`(#n) et `out` #n, ...

Exemple (entrée)

Lecture du périphérique 2 (Console) jusqu'à l'appui sur la touche Escape.

```
do
  i=inp(2)
  ift i=$1b stop
  print /h/ i;
loop
```

Exemple (sortie)

Ecriture sur le périphérique 5 (Vidéo).

```
print /c/
for i=0,255
  out 5,i
  ift modr(i,16)=15 print
next i
stop
```

INP?(k)

OUT?(k)

V_fonctions Etat du périphérique

k

entier, numéro du périphérique

La fonction `inp?` donne l'état du périphérique $k \in [0,3]$ en entrée. La fonction `out?` donne l'état du périphérique $k \in [0,4]$ en sortie. Ces fonctions renvoient -1 si le périphérique est prêt, et 0 sinon.

Exemple

Le programme demande le branchement de l'imprimante (périphérique 0) et attend jusqu'à ce qu'elle soit prête.

```
if out?(0)=0
  print "Brancher l'imprimante"
  do
    ift out?(0) exit
  loop
endif
```

PRINT [#n] [/ {opt} /] liste

WRITE [#n] [/ {opt} /] liste

LPRINT [#n] [/ {opt} /] liste

Commandes Ecriture sur le canal n

n

entier $n \in [0,103]$ numéro de canal de sortie

Nous avons déjà vu les commandes `print` et `lprint`, sans `#n`, qui écrivent sur les périphériques "VBS:" et "LBS:". En fait, la commande `print` (resp `lprint`) sans `#n` utilise le canal 102 (resp 103), c'est à dire le périphérique "vbs:" (resp "lbs:"), à moins que ce périphérique n'ait été redéfini par `print_dev` (resp `lprint_dev`). Ces commandes écrivent sur le canal n, lorsque `#n` est indiqué. On peut ajouter "," ou ";", sans signification particulière, après `#n`.

La commande `write` diffère de `print` par les points suivants. Les séparateurs "," et ";" de la liste sont équivalents. Entre les expressions écrites, il y a sortie, sur le canal n, de ",". Il y a envoi en fin de `write` des octets 13 puis 10. La commande `write` est utile pour préparer un fichier qui sera relu par `input`.

Les périphériques "con:" et "vid:" ne sont pas traités comme "vbs:", mais envoient des données brutes vers le système (Gemdos). Les données ainsi écrites ne peuvent être réutilisées, à la différence de "vbs:".

Le périphérique "lst:" envoie les données vers l'imprimante sans aucune conversion, à la différence de "lbs:".

Exemple

Une autre façon d'écrire sur l'écran, en ouvrant un canal "VBS:".

```
open "o",1,"vbs:"
```

```
write #1"hexa";
write #1,/h/§1000
```

Sortie (30 ms)

```
hexa,  3E8
```

On ne peut pas écrire dans le programme ci-dessus :

```
write#1"hexa"
write #1 /h/§1000
```

Nous avons déjà indiqué que, comme `#` est autorisé dans les noms, il est nécessaire de séparer `#1` du nom qui précède (`out`, `print`, etc.). L'autre incorrection vient de ce que le Basic essaiera de prendre l'expression `1 /h/§1000` pour numéro de canal.

INPUT$(i [, #n])

C_fonction Lecture d'octets

i

entier*32 $i \geq 0$

n

entier $n \in [0, 101]$ (défaut $n = 101$) canal d'entrée

La fonction `input$` lit i octets sur le canal n. Si le périphérique est "vbs:", la lecture est faite sur le périphérique "con:". Les octets lus sont renvoyés sous forme de chaîne.

Exemple

Lecture et affichage de deux octets sur le canal 1.

```
c$="abcd"
open "i",1,"mem:",c$
print input$(2,#1)
```

Sortie (20 ms)

```
ab
```

INPUT [#n] ["texte"] nomi {, nomi}

LINE INPUT [#n] ["texte"] nomc {, nomc}

Commandes Lecture d'expressions et chaînes

n

entier $n \in [0, 101]$ (défaut $n = 101$) canal d'entrée

Les commandes `input` et `line input` lisent des données sur le canal n, comme les mêmes commandes sans `#n` chargent des données à partir du clavier (voir la section Entrée clavier). En fait, les commandes sans `#n` utilisent le canal 101, qui est l'entrée clavier, à moins que ce canal n'ait été redéfini par `input_dev`. On peut écrire, sans signification particulière, après `#n` ou "texte" un séparateur "," ou ";".

Les périphériques "con:" et "vid:" sont traités comme "vbs:" (entrée clavier sur écran pleine page). L'argument "texte" n'est possible que pour ces périphériques.

Lors de l'entrée à partir d'un canal autre que "vbs:", la fin de ligne est marquée par les deux caractères `chr$`(13)&`chr$`(10) (`chr$`(13) seul ne suffit pas et produit l'erreur EOL=End Of Line) ou par le seul caractère `chr$`(0).

Exemple

Les 2 nombres écrits sur une même ligne par `write` dans le fichier T.TST sont relus séparément par `input` et simultanément par `line input`. Le mode d'accès "o" (resp "i") est utilisé pour écrire (resp lire) le fichier, et `seek` positionne le pointeur en début du fichier pour le relire.

```
open "O",#1,"T.TST"
write #1,pi;exp(1)
close #1
open "I",#1,"T.TST"
input #1,u,v
print u;v
seek 1
line input #1,c$
print c$
close #1
```

Sortie (3800 ms)

```
0.3141592654~ E+1  0.2718281828~ E+1
0.3141592654~ E+1,  0.2718281828~ E+1
```

Exemples de fichiers virtuels

Les fichiers virtuels "MEM:" sont commodes lors de la mise au point des programmes, car ils économisent le temps des accès disques. Supposons qu'on désire analyser lexicalement un fichier disque ASCII. On commence par mettre au point le programme avec un fichier "mem:",c$, et seulement une fois le programme terminé et testé, on remplacera "mem:",c$ par le nom du fichier.

L'exemple ci-dessous donne le programme en cours de mise au point. Le fichier est lu caractère par caractère avec `input$` jusqu'à la fin du fichier signalée par `eof`. La fonction `upper1$` est utilisée pour déterminer les lettres, qui sont groupées en mots dans la table `dico`. La fonction `search` indique les mots nouveaux, qui sont rangés par `sort` suivant la relation d'ordre `cmp1`. La valeur `F`(i) donne le nombre d'occurrences du mot `dico`(i).

La sortie, pour le moment, est effectuée sur l'écran. Pour programme final, on remplacera "vbs:" par le nom du fichier de sortie. La liste des mots est présentée dans l'ordre alphabétique, puis suivant les fréquences, après un nouveau tri. La commande `permute` assure que pour chaque valeur de la fréquence les mots sont également dans l'ordre alphabétique.

Pour chiffrer le gain en temps de la méthode préconisée, il suffit de comparer le temps d'exécution sans disques (3s) à celui avec disques (≈ 60 s), et de compter le nombre d'exécutions (peut-être une douzaine) qui ont été nécessaires pour mettre en forme le programme.

```
c$="Et puis vinrent les neiges, "
```

```
cadd c$,"les premières neiges de l'absence, sur "
cadd c$,"les grands lés tissés du songe et du réel;"
char dico(100)
index P(100),F(100),i,N,W
clear timer
open "i",1,"mem:",c$
lettres$=cset$(["A","Z"],"AE","OE")
do
  ift eof(1) exit
  dico(N+1)=
  do
    ift eof(1) exit
    a$=input$(1,1)
    ift upper1$(a$) not in lettres$ exit
    cadd dico(N+1),a$
  loop
  if dico(N+1)<>""
    W=W+1
    i=search(dico(1),N+1,1,P(1),cmp1)
    if dico(P(i))=dico(N+1)
      F(P(i))=F(P(i))+1
    else
      N=N+1
      sort dico(1),N,1,P(1),N-1,cmp1
      F(N)=1
    endif
  endif
loop
close #1
print W;" mots    ";N;" entrées    mtimer=";mtimer
if sure?("insérer la disquette|sortie alphabétique")
  open "o",#1,"vbs:"
  for i=1,N
    print #1;dico(P(i));f;
  next i
  close #1
endif
print æ;"Tri en fréquence"
permute dico(1),N,1,P(1)
permute F(1),N,1,P(1)
sort F(1),N,1,P(1),0
if sure?("insérer la disquette|sortie fréquences")
  open "o",#1,"vbs:"
  for i=1,N
```

```
      ift modr(i,100)=0 print i;" écriture  mtimer=";mtim
       er
      print #1;dico(P(i));f;justl$(F(P(i)))fff;
    next i
    close #1
  endif
```

Sortie

```
 21 mots      17 entrées    mtimer=  2955
absence de du Et et grands l les lés neiges premières puis réel songe
 sur tissés vinrent
Tri en fréquence
absence 1   de 1   Et 1   et 1   grands 1   l 1   lés 1   premières 1
   puis 1   réel 1   songe 1   sur 1   tissés 1   vinrent 1   du 2   n
 eiges 2   les 3
```

Une autre application des fichiers virtuels consiste en leur utilisation comme tampon d'entrée ou de sortie de fichiers disques, ce qui permet d'accélérer les opérations disques. L'exemple suivant, qui détermine la liste des caractères du fichier “help.z” est basé sur ce principe. Il utilise les commandes et fonctions `open`, `close`, `lof`, `loc`, `seek`, `inp` et `input$`.

```
  index CAR(255)
  char C
  OPEN "I",#0,"HELP.Z"
  OPEN "I",#1,"MEM:",C
  DO
    L=LOF(0)-LOC(0)
    IFT L<=0 EXIT
    C=INPUT$(MIN(L,10240),0)
    SEEK #1,0
    FOR I=1,LEN(C)
      CAR(INP(#1))=-1
    next I
    PRT
  LOOP
  CLOSE
  stop
PRT:print /CH/"     ";CONC$(J=0,$FF,$10 OF justr$(RIGHT$(J,
 2),3))
  FOR I=0,15
    print /H/justl$(I,3);"  ";CONC$(J=0,$FF,$10 OF justr$
     (CAR(I+J),3))
  next I
  print "L=";LOC(0),"timer=";timer;
  return
```

FIELD [#] n {, fld}

Commande Définition des champs *fld* de l'enregistrement

GET #n [, r]

Commande Lecture de l'enregistrement *r*

PUT #n [, r]

Commande Ecriture de l'enregistrement *r*

n

entier $n \in [0, 100]$ numéro de canal de type "R"

r

entier*32 > 0

Par défaut, $r = 1$ pour la première commande `put` ou `get`, et la valeur par défaut de r est ensuite incrémentée de 1 à chaque nouvel appel.

fld

```
 a AS c$
|a AS i
```

a

entier*32 > 0, longueur du champ

c$

nomi de type char

i

nomi d'index**size*

Chaque *fld* fixe un champ de longueur a. La somme des longueurs des champs de l'enregistrement d'un fichier "R" doit être égale à la longueur spécifiée dans la commande `open`. L'écriture :

```
field 1, 4 as i
field 1, 12 as d$
```

équivaut à :

```
field #1, 4 as i, 12 as d$
```

La spécification `a as c`, où `c` est un nomi de type char, produit lors de l'exécution de la commande `field` une instruction implicite :

```
c=space$(a)
```

qui remplit `c` avec a espaces. Notons aussi que si `c` était un nom inconnu, il lui est attribué par la commande `field` le type char.

Lors de la commande `put` la longueur de chaque champ doit être celle définie par `field`, sous peine de sortie erreur. Il est donc recommandé de modifier les variables de type char des champs seulement par les commandes `lset`, `rset` et `mid$`.

La spécification `a as i` où `i` est un nomi d'index définit le champ de a octets à partir de l'adresse `ptr`(i). Ce champ n'est pas modifié lors de la commande `field`, à la différence des champs de type char. Dans :

```
index L(100)
field #1, 400 as L(1)
```

le champ correspond aux index L(1) à L(100), dont chacun occupe 4 octets. Noter que la donnée :

```
field #1, 401 as L(1)
```

produit une erreur Hors Du Tableau.

Exemple

On ouvre en accès sélectif le fichier virtuel “mem:” D$. Chaque enregistrement comporte deux champs, une variable de type char de 16 octets et un index (4 octets). La longueur d'enregistrement, précisée dans `open` est 20. Le champ de type char est rempli par `lset`. Si on désire ensuite utiliser un fichier disque, on remplacera “MEM:”,D$ dans `open` par le nom du fichier.

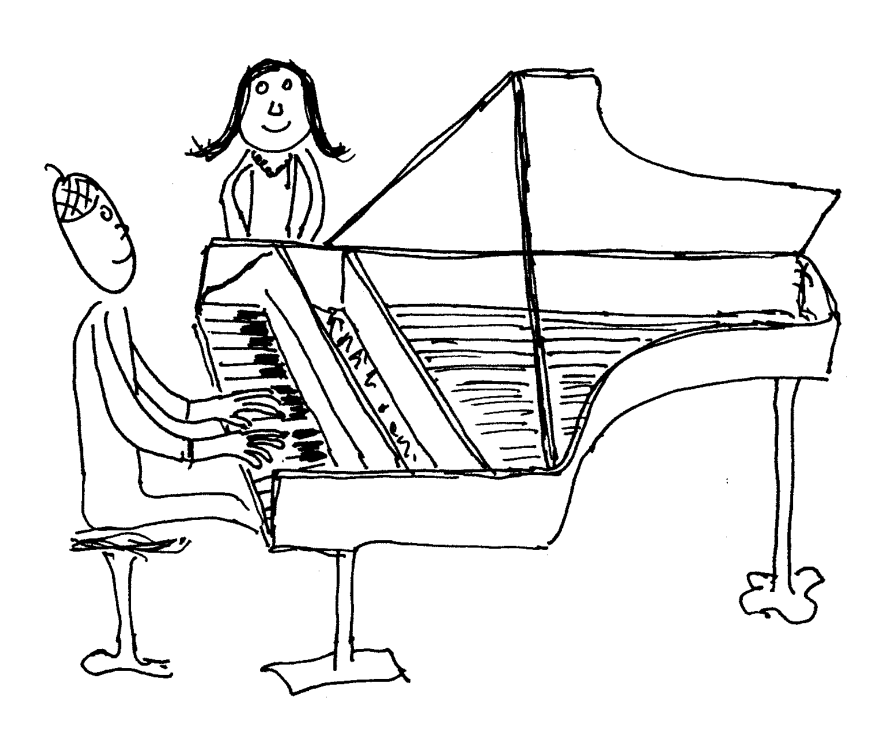
```
Index TEL
OPEN "R",1,"MEM:",D$,20
FIELD #1,16 AS NOM$
FIELD #1, 4 AS TEL
TEL=60779250
lset NOM$="Préfecture"
PUT #1
TEL=46277057
lset NOM$="SEFRANE"
PUT #1
for i=1,2
  GET #1,i
  print NOM$;using " ## ## ## ##",TEL
next i
close #1
```

Sortie (115 ms)

```
Préfecture       60 77 92 50
SEFRANE          46 27 70 57
```

11

Programmation



Structures IF

IF x1

bloc1

ELSE IF x2

bloc2

...

ELSE

blocn

ENDIF

Commandes If structuré

x1, x2, ...

expr

bloc1, bloc2, ...

Désignent des lignes d'instructions Basic.

Variantes d'écriture

On peut écrire `if x1 then` au lieu de `if x1`, et `else x2` au lieu de `else if x2`.

Le nombre de blocs de la structure est illimité. Voici les structures les plus simples :

```
IF x1
  bloc1
ENDIF
```

et :

```
IF x1
  bloc1
ELSE
  bloc2
ENDIF
```

Si `x1` $\neq 0$ les lignes bloc1 sont exécutées, mais bloc2 ne sera pas exécuté et l'exécution continue après la ligne de la commande `endif`. Si `x1` $= 0$ on saute le bloc1 et on exécute le bloc2, s'il est donné (cas avec `else`). Les blocs eux aussi peuvent contenir un nombre illimité de structures `if`.

Dans les exemples :

```
IF x1
  bloc1
ELSE IF x2
```

```
  bloc2
ENDIF
```

et :

```
IF x1
  bloc1
ELSE IF x2
  bloc2
ELSE
  bloc3
ENDIF
```

si `x1` $\neq 0$ seul bloc1 est exécuté.
si `x1` $= 0$ et `x2` $\neq 0$ seul bloc2 est exécuté.
si `x1` et `x2` sont égaux à 0 seul bloc3 est exécuté (deuxième exemple).

Il est totalement correct de sortir d'un bloc du `if` par une instruction de branchement quelconque. On pourra ainsi quitter un bloc par `goto`, `return`, `exit`, `exitif`, etc. Il est également correct d'effectuer un branchement vers un bloc du `if`. Dans ce cas aussi, après la fin du bloc, l'exécution se poursuit après le `endif` de la structure.

IFT expr [THEN] instruction

Commande IF uniligne

Dans le mot clef `ift` le `t` rappelle le `then` pour distinguer cette commande de la structure `if` ... `else` ... `endif`. Si expr a une valeur différente de zéro, l'instruction après `then` est exécutée. L'instruction peut être n'importe quelle commande du Basic 1000d ou être omise (utile pour les fonctions comme `gemdos` dont on ne veut pas la valeur, ou pour rectifier la pile par `ift pop`).

Exemple

```
a=random(2)
ift a=0 print "a est nul"
ift a=1 print "a vaut un"
```

Sortie (25 ms)

```
a est nul
```

On peut presque toujours omettre `then`. Les procédures et variables dont le nom (par exemple e1), commence par E suivi d'un chiffre 0–9 peuvent provoquer des erreurs dans la commande `ift`.

Exemple

```
var x,e1
ift x=0 e1=1
```

Sortie (15 ms)

```
*ERREUR* INSTRUCTION ILLEGALE
ift x=0 e1?
2.ift x=0 e1=1
```

L'erreur provient du fait que "0 e1" a été décodé comme un nombre, en notation exponentielle. On écrira pour éviter cette erreur :

```
ift (x=0) e1=1
```

ou :

```
ift x=0 then e1=1
```

Structures SELECT CASE

SELECT CASE x

CASE dcomp1

bloc1

CASE dcomp2

bloc2

...

CASE OTHERS

blocn

ENDSELECT

Commandes Structure select case

x

expr ou exprchaîne

dcomp1, dcomp2, ...

membres de droite des comparaisons ou inclusions. Si on juxtapose `x` et dcomp1, on doit obtenir une exprc qui ne soit pas une expra.

Variante d'écriture

On peut écrire `select x` au lieu de `select case x`.
Dans la structure `select case`, au plus un des blocs (`bloc1` ou `bloc2` ou ...) est exécuté. C'est le premier `bloci` tel que la condition `x dcondi` soit vraie, ou bien le bloc après `case others` si les conditions précédentes étaient toutes fausses. L'expr ou exprchaîne `x` n'est évaluée qu'une seule fois.

Exemple

Suivant la valeur de `random`(100), un et un seul des `print` est exécuté, puis l'exécution continue après la ligne de la commande `endselect`. Par exemple pour la valeur 40, seul "beaucoup" est imprimé, le `print` suivant est ignoré.

```
SELECT CASE random(100)
CASE <20
```

```
  print "un peu"
CASE IN [20,40]
  print "beaucoup"
CASE IN (40,[70,100,2])
  print "à la folie"
CASE OTHERS
  print "pas du tout"
ENDSELECT
```

Sortie (35 ms)

```
beaucoup
```

Exemple

Attend l'entrée d'un nom au clavier, puis le teste à l'aide d'une structure `select case`.

```
char c
input "Entrer un nom",c
nodistingo
select case c
case in ["A","KINKAJOU"[
  print "tome 1"
case in ["KINKAJOU","ZYTHUM"]
  print "tome 2"
case others
  print "autre"
endselect
```

Comme dans le cas des blocs de la structure `if`, la sortie et l'entrée dans un bloc de la structure `select` est totalement libre.

ON x instr_1, instr_2, . . . , instr_n

Commande Sélection d'une instruction

x

entier*32

instr_i

instruction en Basic ($i = 1, \ldots, n$)

L'instruction instr_i ne peut pas contenir de virgule non parenthésée ou non entre guillemets. Il est cependant admis, s'il n'y a pas de guillemets dans instr_i, de l'entourer de guillemets.

Si $x \in [1, n]$ alors la commande `on` exécute la x-ième instruction instr_x et passe à la ligne suivante (sauf si instr_x est un branchement). Sinon, la commande passe simplement à la ligne suivante. La commande `on` x . . . se comporte donc comme un `select case` uniligne et à sélecteur entier.

Exemple

```
for x=0,10
```

```
          print "x=";x
          on x print "1","print x,2",,x=min(x+1,10),,stop
        next x
```

Sortie (205 ms)

```
x=  0
x=  1
1
x=  2
  2              2
x=  3
x=  4
x=  6
```

Noter que si Sp1, Sp2(w), ... sont des procédures

```
on x gosub Sp1, Sp2(w), ...
```

appelle les procédures Sp1 (resp Sp2(w), ...) si $x = 1$ (resp $x = 2, \ldots$). En effet `gosub` est facultatif devant le nom d'une procédure. Par contre,

```
on x goto L1, L2
```

ne se comporte pas comme dans d'autres Basics. Si $x = 1$, il y a branchement en L1. Par contre, si $x = 2$, l'instruction

```
L2
```

est exécutée, c'est à dire qu'il y a appel de la procédure L2. Si on voulait effectuer un branchement à L2, il fallait répéter `goto` :

```
on x goto L1, goto L2
```

Boucles FOR

FOR i=a TO b [STEP c]

Bloc

NEXT [i {, j}]

Commandes de boucle FOR

i

index*32

a, b, c

entiers*32 $c \neq 0$

Variante d'écriture

Les mots clefs `to` et `step` peuvent être remplacés par des virgules.

Le bloc de lignes entre les commandes `for` et `next` est exécuté au moins une fois avec l'index `i` prenant successivement toutes les valeurs a, $a+c$, $a+2c$, …, $a+nc$ comprises entre a et b inclus. Si le pas de la boucle c est omis il est pris égal à 1 (si $a \leq b$) ou à -1 (si $a > b$).

Exemple

```
for J=0,8,2
  W=X^J
  PRINT J;W
next
```

Sortie (115 ms)

```
0  1
2  X^2
4  X^4
6  X^6
8  X^8
```

Boucles imbriquées

Le nombre d'imbrications est limité par la variable d'état `s_pro` (qui peut être modifiée si nécessaire). Le `next` peut être commun si on indique les index de boucle.

Exemple

La boucle `I` correspond implicitement au pas $c = -1$.

```
for I=7,6
  for J=1 to 3
    print "  (";I;J;")";
next J,I
```

Sortie (120 ms)

```
(  7  1)  (  7  2)  (  7  3)  (  6  1)  (  6  2)  (  6  3)
```

Exemple anormal

Dans cet exemple, par suite de l'absence de `next J`, il n'y a pas vraiment de boucle sur `J`, qui reste toujours égal à 1.

```
for I=1,2
A:print "Ligne A > I=";I
  for J=1,18
  B:print "Ligne B > I=";I;", J=";J
  next I
```

Sortie (120 ms)

```
Ligne A > I=  1
Ligne B > I=  1, J=  1
Ligne A > I=  2
Ligne B > I=  2, J=  1
```

On peut utiliser plusieurs `next` pour la même boucle, ainsi

```
for i=1,10
```

```
    ift i=2 next i
    ift i=3 next i
  next i
```

est correct.

Explications

La commande `next I,J, ...` équivaut à la suite :

```
next I
next J
...
```

La commande `next I` termine toutes les boucles commencées après la boucle `I`, puis ajoute c à I et si la valeur obtenue est entre a et b reprend la boucle. Sinon elle termine la boucle `I`. La commande `next` sans index concerne la dernière boucle.

La commande `for I=`..., lorsque le programme est déjà dans une boucle sur `I` non terminée, termine toutes les boucles commencées après l'ancienne boucle `I`.

Dans :

```
b=10
c=1
for i=1,b,c
  b=b+1
  c=c+1
next i
```

les modifications de la borne b et du pas c de la boucle ne changent pas la boucle. Les limites et pas sont calculés une fois pour toutes lors de la commande `for`. On peut en particulier utiliser sans crainte :

```
for i=1,i
next i
```

Dans :

```
for i=1,10
  i=i-1
next i
```

l'index de boucle est modifié. La boucle, ici, en devient sans fin.

Exemple

Illustration de la sortie de boucles `for`. Le programme recherche des nombres premiers qui, en système décimal, possèdent la propriété suivante :
Si on modifie un seul de ses chiffres, on n'obtient jamais un nombre premier. Le premier chiffre n'est pas changé en 0.

On effectue une boucle (qui n'est pas une boucle `for`, mais réalisée à l'aide de `goto`) sur tous les nombres premiers, `r`, tels qu'en changeant le chiffre des unités on obtienne un nombre non premier. Cette boucle est incrémentée (lignes 2–5) en prenant `r` = `prime(d)` où `d` est un multiple de 10, et en vérifiant que le nombre premier `r1` suivant `r` n'est pas dans la même dizaine que `r`. On

détermine ensuite les nombres `r2` obtenus en changeant le `k`-ième chiffre de `r` à partir de la droite (`k` va de 2 à `m`, le nombre de chiffres de `r`) par le chiffre `i` (de 0 à 9). Noter que le chiffre de gauche de `r` (pour `k` = `m`) n'est pas changé en `i` = 0. Cela est réalisé en donnant pour borne inférieure de la boucle `for` sur `i` l'expression `-(k=m)` qui vaut 0 ou 1. Si un de ces nombres est premier, on sort des boucles `for` sur `i` et `k` et on poursuit la boucle sur `r`. La sortie des boucles `for` s'effectue par un simple `goto`. Lorsque le programme exécute ensuite :

```
for k=2,m
```

les anciennes boucles sur `k` et `i` sont dépilées. Noter que l'instruction :

```
ift r2<>r ift prtst(r2) goto a
```

est équivalente à :

```
ift r2<>r and prtst(r2) goto a
```

mais plus rapide, parce que le calcul de `prtst`(`r2`) (qui est très long) est évité lorsque `r2` = `r`.

```
   r=0
 a:d=divr(r,10)*10+10
   r=prime(d)
   r1=prime(r+1)
   ift divr(r1,10)=divr(r,10) goto a
   m=gint(log10(r))
   for k=2,m
     f=10^(k-1)
     r1=modr(r,10^k)
     r1=divr(r1,f)*f
     r1=r-r1
     for i=-(k=m),9
       r2=r1+f*i
       ift r2<>r ift prtst(r2) goto a
   next i,k
   print r;"  (timer=";timer;")"
   goto a
```

Sortie

```
294001  (timer=  7685)
505447  (timer=  15016)
584141  (timer=  17902)
...
```

NONEXT i

Commande Termine une boucle FOR

i

index

La commande `nonext i` termine la boucle `for` sur `i` et les boucles ouvertes après cette boucle i.

Exemple

Si on supprime `nonext i`, au moment où on commence la deuxième boucle en `i`, la première boucle n'est pas terminée. Il en résulte qu'on termine les deux boucles (`i` et `j`) avant de commencer cette boucle, ce qui provoque une erreur sur `next j`.

```
for i=1,1000
  nonext i
  for j=1,10
    for i=1,1
    next i
  next j
```

PROD({bouclei} OF p)

V_fonction Produit sur les boucles

SUM({bouclei} OF p)

V_fonction Somme sur les boucles

CONC$({bouclei} OF s)

C_fonction Concaténation sur les boucles

bouclei

représente une commande `for` complète. Cependant le premier mot clef `for` peut être omis.

p

expr

s

exprchaîne

Les fonctions `sum`, `prod` et `conc$` permettent d'effectuer simplement des sommes, produits et concaténations. Ainsi, pour calculer la somme S des éléments d'un tableau T(10,10), on peut utiliser :

```
S=sum(for i=0,10 for j=0,10 of T(i,j))
```

qui revient à exécuter le programme :

```
S=0
for i=0,10
  for j=0,10
    S=S+T(i,j)
  next j
next i
```

Exemple

Calcul de la somme $1+2+\cdots+10$, puis du produit des nombres impairs de 1 à 33.

```
print sum(FOR I=1,10 OF I)
print prod(ia=1,33,2 of ia)
```

Sortie (80 ms)

```
55
6332659870762850625
```

Exemple

Ecriture de 10 lignes identiques, puis de la table des symboles ASCII.

```
print conc$(I=1,10 of  "pour continuer appuyer sur une
 touche" & chr$(13))
ift keyget
print conc$(i=0,15 of conc$(j=i,$ff,16 of f&chrp$(j))a)
```

Les boucles des fonctions `sum`, `prod` et `conc$` peuvent entrer en conflit avec les boucles des véritables commandes `for` ... `next`. Cela peut être la cause d'erreurs comme dans l'exemple suivant :

```
for I=1,100
  print sum(I=1,10 OF I)
next
```

En sortant de la commande `sum`, `I` vaut toujours 11, et il s'en suit un bouclage infini.

Exponentiation par PROD et intégration par SUM

Pour n grand, le calcul de $w = (1 + x)^n$ est plus rapide par :

```
w=PROD(J=1,n OF 1+x)
```

que par `w=(1+x)^n`. De même, le calcul de l'intégrale en x du polynôme w est plus rapide par

```
Iw=SUM(J=ORD(w,x)+1,DEG(w,x)+1 OF COEF(w,x,J-1)/J*x^J)
```

que par `Iw=INTG(w,x)`. Voici les temps (en s) pour divers n. Le ? correspond à une erreur mémoire (1040ST).

n	prod	(1+x)^n	sum	intg
128	4	3	2	5
512	105	205	66	95
1024	639	2492	299	?

Le calcul par `(1+x)^512` est effectué par élévations successives au carré, c'est à dire seulement par quelques produits de grands facteurs. La fonction `prod` qui effectue un grand nombre de produits, mais avec toujours le petit facteur $1 + x$, se trouve être beaucoup plus rapide. Notons toutefois que pour $n \leq 212$, c'est `(1+x)^n` qui est le plus rapide.

La fonction `sum` est plus rapide et utilise moins de mémoire que `intg`, parce que `intg` travaille sous forme factorisée.

Boucles FORV et FORC

FORV v IN v_ensemble

Blocv

NEXTV

Commandes de boucle FORV

v

nomi de type var

Dans la commande `forv`, le v_ensemble doit être de type fini, c'est à dire consister d'expr isolées et de progressions. Les segments continus ne sont pas autorisés. Le bloc de lignes Blocv est exécuté plusieurs fois, `v` parcourant l'ensemble des valeurs du v_ensemble. Si le v_ensemble est vide, Blocv n'est pas exécuté et l'exécution continue après la commande `nextv`.

Exemple

Le v_ensemble peut comporter la même valeur plusieurs fois, comme ici trois fois la valeur 1. La boucle est alors exécutée trois fois avec cette valeur. Le v_ensemble est calculé au moment de l'instruction `forv` une fois pour toutes. La modification de `a` ne change rien à la boucle. La modification de la variable de boucle `v`, à l'intérieur de la boucle ne modifie pas non plus la boucle, à la différence des boucles `for`.

```
a=100
forv v in (1,1,[1,a,25/2])
  print v;
  a=7
  v=7
nextv
```

Sortie (155 ms)

```
1  1  1  27/2  26  77/2  51  127/2  76  177/2
```

FORC c IN c_ensemble

Bloc

NEXTC

Commandes de boucle FORC

c

nomi de type char

Dans la commande `forc`, le c_ensemble doit consister d'exprchaînes isolées et de csegments $[a, b]$ où a et b sont de longueur 1. Les autres csegments ne sont

pas autorisés. La boucle est l'analogue de la boucle `forv` pour `c` parcourant l'ensemble des chaînes du c_ensemble.

Exemple

```
        forc c in ("alpha",["A","G"])
          print c;" ";
        nextc
```

Sortie (60 ms)

```
  alpha A B C D E F G
```

Boucles DO, WHILE et REPEAT

DO

bloc

LOOP

Commandes de boucle DO

Répète le bloc d'instructions entre les commandes `do` et `loop`. Sortie par la commande `exit`, ou `return` dans un sous-programme.

Exemple

Le retour de la procédure `attend` n'a lieu que lorsque un octet est disponible sur l'entrée série.

```
        attend
        print inp(1)
        stop
  attend:do
          ift inp?(1) return
        loop
```

REPEAT

bloc

UNTIL expr

Commandes de boucle REPEAT

Répète le bloc d'instructions entre les commandes `repeat` et `until`. Sortie en fin de bloc si expr est non nul, ou dans le bloc par `exit`.

Exemple

Il y a sortie de la boucle lorsque x prend la valeur 0.

```
        x=10
        repeat
```

```
    print x;
    x=x-1
  until x=0
```

Sortie (140 ms)

```
10  9  8  7  6  5  4  3  2  1
```

WHILE expr

bloc

WEND

Commandes de boucle WHILE

L'exécution du bloc entre les commandes `while` et `wend` est répétée tant que expr est non nul. Si expr est nul au premier passage, le bloc est sauté. La sortie du bloc est possible par `exit`.

Exemple

La boucle n'est plus répétée lorsque x prend la valeur 0.

```
  x=10
  while x
    print x;
    x=x-1
  wend
```

Sortie (130 ms)

```
10  9  8  7  6  5  4  3  2  1
```

Sortie des structures

EXIT

Commande Sortie de boucle (`for`, `forv`, `forc`, `do`, `repeat`, `while`)

La commande `exit` fait sortir de la dernière boucle seulement. L'exécution se poursuit alors après le `next`, `nextv`, `nextc`, `loop`, `until` ou `wend` correspondant.

Exemple

La commande `exit` termine la boucle `forv` interne lorsque u=v.

```
  forv v in [1,2,1]
    forv u in (7/2,1,2)
      ift u=v exit
      print v;u
    nextv
  nextv
```

Sortie (105 ms)

```
1  7/2
2  7/2
2  1
```

Pour que la commande fonctionne bien il faut que `exit` soit dans le bloc de la boucle. Ainsi,

```
        do
          P
        loop
        stop
      P:exit
```

n'est pas correct. De plus, les fins de boucles doivent être apparentes. Ainsi,

```
        for i=1,10
          exit
          ift 1 next i
```

est incorrect (`next i` caché) de même que :

```
        for i=1,10
          exit
          for j=1,10
        next j,i
```

(ne pas utiliser de `next` à plusieurs index avec `exit`).

La commande `exit` peut se trouver dans des structures `if` et `select` incluses dans une boucle. Elle provoque bien la sortie de la boucle, les structures `if` et `select` étant invisibles pour la commande `exit`.

Exemple

La commande `exit` fait continuer l'exécution après `nextv`.

```
        forv i in (1,4,6,10)
          if i=4
            exit
          endif
          print i
        nextv
```

Sortie (30 ms)

```
1
```

Exemple

Le programme fait deviner un entier. La commande `exit`, située dans une structure `select case`, termine la boucle `do`.

```
        print "Devinez l'entier x"
        x=random(100)+1
        do
          input y
          select y
          case =x
            exit
```

```
      case <x
        print y;" < x"
      case >x
        print y;" > x"
      endselect
    loop
    print "Trouvé"
```

Exemple de dialogue

```
Devinez l'entier x
INPUT >
75
  75 > x
INPUT >
55
  55 < x
INPUT >
57
Trouvé
```

EXITIF

Commande Sortie d'une structure IF

Après `exitif`, l'exécution se poursuit après le endif de la structure.

Exemple

Après `exitif`, l'exécution continue sur la ligne c, en non en b.

```
    for i=1,10
      if i<4
        ift i=1 exitif
        if i=3
          print "a";i
        endif
      b:print "b";i
      endif
   c:next i
```

Sortie (100 ms)

```
b  2
a  3
b  3
```

EXITSELECT

Commande Sortie d'une structure SELECT

Après `exitselect` l'exécution se poursuit après le `endselect` de la structure.

Exemple

```
    for i=1,10
      select case i
```

```
    case <4
      if i=2
        exitselect
      endif
      print "b";i
    endselect
  next i
```

Sortie (90 ms)

```
b  1
b  3
```

Il est incorrect, en général, de placer `exitif` et `exitselect` dans une boucle inclue dans la structure. En effet ces commandes ne terminent pas les boucles.

Exemple

Après `exitif` l'exécution continue ligne a, mais aucune boucle n'a été fermée. La commande `nextv` renvoie donc ligne b, après avoir modifié la variable u. Une fois la boucle en u terminée, le programme arrive en fin de source sans avoir bouclé sur la variable v.

```
forv v in [1,5,1/2]
  if v=1
    forv u in [-7,-8,-1/3]
    b:exitif
    nextv
  endif
a:print v;u
nextv
```

Sortie (115 ms)

```
1 -7
1 -22/3
1 -23/3
1 -8
```

Remarques sur les structures

Basic 1000d est assez laxiste sur la donnée des boucles `for`, on pourra ainsi donner aucune, une ou plusieurs fins (qui peuvent être cachées dans des commandes `xqt` ou `ift`) à la même boucle. Mais attention ne pas oublier qu'une boucle `for i=`... commencée avec un index de boucle non terminée, termine toutes les boucles après la première boucle `i`.

Les commandes `nextv`, `nextc`, `wend`, `until` et `loop` terminent les boucles `for` commencées dans leur bloc de répétition. Par contre, par exemple, `wend` ne peut pas terminer une boucle `forv` commencée et non fermée dans son bloc de répétition. On obtient alors l'erreur Mauvaise Imbrication.

Les commandes de structures (`if` et `select case`) et de boucles (`do`, `repeat`, `while`, `forv` et `forc`) doivent être visibles, c'est à dire que des instructions comme :

```
ift x else
xqt "wend"
```

dans lesquelles `else` et `wend` sont cachés sont incorrectes. En effet lorsque `while` voudra sauter le bloc, il sera incapable de trouver `wend` dans `xqt`. De même le `if` précédant le `else` caché sera incapable de le localiser.

Traitement des chaînes

MIN$(s1, s2 {, si})

MAX$(s1, s2 {, si})

C_fonctions Minimum ou maximum

s1, s2, si

exprchaînes

Les fonctions `min$` et `max$` distinguent ou non les majuscules des minuscules suivant la variable d'état `distingo` (ou `nodistingo`). Les noms des fonctions analogues pour les réels ont des noms différents, `min` et `max`.

Exemple

```
c$="maj"
d$="MIN"
print min$(c$,d$);distingo
nodistingo
print min$(c$,d$);distingo
```

Sortie (35 ms)

```
MIN  0
maj  1
```

UPPER$(s)

UPPER1$(s)

C_fonctions Conversion en majuscules

LOWER$(s)

C_fonction Conversion en minuscules

s

exprchaîne

Avec `upper$`, les lettres non accentuées a–z sont changées en A–Z, mais pas les lettres é, è, etc. La fonction `upper1$` supprime les signes diacritiques des lettres a–z, change toutes les lettres en majuscules et défait les ligatures œ et æ.

La fonction `lower$` change les majuscules A–Z en minuscules.

Exemple

L'appel d'`upper$` puis de `lower$` ne redonne pas la chaîne de départ.

```
M$=upper$(" ** Bibliothèque Mathématique ** ")
print M$a;lower$(M$)
print upper1$("fœhn, sèche-cheveux")
```

Sortie (95 ms)

```
 ** BIBLIOTHèQUE MATHéMATIQUE **
 ** bibliothèque mathématique **
FOEHN, SECHE-CHEVEUX
```

CHANGE$(s {, ui, vi})

C_fonction Substitutions de chaîne

s, ui, vi

exprchaîne

La fonction `change$` remplace dans la chaîne `s` toutes les occurrences de la chaîne `u1` par la chaîne `v1`, puis sur le résultat change `u2` en `v2`, ... Dans `u1`, `u2`, ... on distingue ou non les majuscules des minuscules selon la variable d'état `distingo` (ou `nodistingo`).

Exemple

La lettre `a` est changée en `ab`, puis la lettre `b` en `c`.

```
print change$("abab","a","ab","b","c")
```

Sortie (15 ms)

```
accacc
```

CHANGES$(s ,[-] sp)

C_fonction Substitutions de caractères

s

exprchaîne à modifier

sp

exprchaîne (au plus les 256 premiers caractères sont pris en compte)

La fonction `changes$` remplace chaque code ASCII de la chaîne `s` par un autre, sans changer la longueur de `s`. Chaque code ASCII $i \in [0, 255]$ devient le $(i+1)$-ième caractère de la chaîne `sp`, ou $FF si $i \geq$ `len(sp)`.

Lorsque le signe − précède `sp`, le code i est transformé en j si i est le $(j+1)$-ième caractère de la chaîne `sp`. Ainsi, lorsque `sp` est formé d'une permutation des 256 caractères distincts 0 .. 255 les transformations :

```
t=changes$(s,sp)
s=changes$(t,-sp)
```

sont inverses l'une de l'autre.

Exemple 2001

```
char sp
sp=chr$(255)&conc$(i=0,254 of chr$(i))
print changes$("IBM",sp)
print changes$("HAL",-sp)
```

Sortie (500 ms)

```
HAL
IBM
```

JUSTR$(s [, a [, f]])

JUSTL$(s [, a [, f]])

JUSTC$(s [, a [, f [, g]]])

C_fonctions Justification à droite, gauche et centrage

s

exprchaîne

a

entier*32 (défaut $a = 0$)

f, g

entier $\in [0, 255]$ (défaut $g = f$ et $f = 32$)

Appelons *sp* la chaîne obtenue en enlevant les espaces aux extrémités de la chaîne *s*. Ces C_fonctions créent une chaîne de longueur `max`(`len`(*sp*), *a*) en rajoutant des caractères de code ASCII *f* et *g* (par défaut des espaces) :

`justr$` devant la chaîne *sp*

`justl$` après la chaîne *sp*

`justc$` devant (*f*) et après (*g*) *sp* de façon (presque) égale.

Les formes comme `justl$`(*s*), avec seulement le premier argument, suppriment les espaces aux extrémités de la chaîne *s*.

Exemple

La fonction `justr$` utilisée avec $f = 48$ permet d'écrire des nombres précédés de zéros.

```
print justc$(justr$(2^20,15,$30),30)
print justc$(chr$($2d,15),30,$3c,$3e)
```

Sortie (85 ms)

```
         000000001048576
<<<<<<<--------------->>>>>>>>
```

LEFT$(s [, a])

RIGHT$(s [, a])

MID$(s, d [, a])

C_fonctions Troncation

s

exprchaîne

a

entier*32

d

entier*32 $d > 0$

La fonction `left$` (respectivement `right$`) renvoie les a (défaut $a = 1$) caractères de gauche (respectivement de droite) de la chaîne s. La fonction `mid$` renvoie les a caractères à partir du d-ième caractère de la chaîne s. Si a est omis ou négatif ou très grand, `mid$` renvoie toute la chaîne s à partir du d-ième caractère.

Exemple

```
c$="abcdefg"
print left$(c$)
print right$(c$,3)
print mid$(c$,3,2)
print mid$(c$,3)
```

Sortie (50 ms)

```
a
efg
cd
cdefg
```

LSET c$=s

RSET c$=s

MID$(c$, d [, a])=s

Commandes Insertion de s dans `c$`

c$

nomi de type char

s

virchaîne

a

entier*32 (Par défaut $a = -1$)

d

entier*32 $d > 0$

Ces commandes modifient la variable `c$`, sans modifier sa longueur. Leur emploi est recommandé en conjonction avec `field`, pour créer les champs des fichiers à accès sélectif. La commande `lset` (resp `rset`) place s à gauche (resp à droite) de `c$` et complète avec des espaces. Si s est plus long que `c$`, seulement les `len(c$)` premiers caractères de s sont utilisés. Ainsi les deux instructions suivantes ont le même effet :

```
lset c$=s
c$=justl$(left$(s,len(c$)),len(c$))
```

La commande `mid$` place s dans `c$` à partir du d-ième caractère. Le nombre de caractères insérés est limité à a caractères ($a < 0$ correspond à une valeur infinie). Les autres caractères de `c$`, avant le d-ième et après le dernier caractère inséré, ne sont pas modifiés.

Exemple

A l'aide de la procédure `ra`, on écrit l'instruction d'insertion, et la chaîne avant et après l'insertion. L'exécution des commandes `lset`, `rset` et `mid$` à lieu sur la première ligne de la procédure `ra`.

```
    u$="123456789"
    c$=u$
    ra lset c$="abc"
    ra rset c$="efg"
    ra rset c$="abcdefghijklm"
    ra mid$(c$,4,2)="abc"
    stop
ra:@1
    print a;arg$(1)
    print ">";u$
    print ">";c$;"<"
    c$=u$
    return
```

Sortie (245 ms)

```
lset c$="abc"
>123456789
>abc      <
rset c$="efg"
>123456789
>      efg<
rset c$="abcdefghijklm"
>123456789
>abcdefghi<
```

```
mid$(c$,4,2)="abc"
>123456789
>123ab6789<
```

MIRROR$(s)

C_fonction Chaîne lue à l'envers

s

exprchaîne

La fonction `mirror$` renverse la chaîne *s*.

Exemple

Il faudrait retourner les caractères pour plus d'exactitude.

```
print mirror$("Le boustrophédon est")
print "une façon   d'écrire"
print mirror$("alternativement   de")
print "gauche  à  droite et"
print mirror$("de droite à gauche.")
```

Sortie (125 ms)

```
tse nodéhportsuob eL
une façon   d'écrire
ed   tnemevitanretla
gauche  à  droite et
.ehcuag à etiord ed
```

CMP(s, sp)

CMP1(s, sp)

V_fonctions Comparaison de deux chaînes

s, sp

exprchaîne

La table suivante donne la valeur renvoyée par la fonction `cmp`(*s*, *sp*) suivant l'ordre lexicographique des chaînes *s* et *sp*. Cet ordre dépend de la variable d'état `distingo` (ou `nodistingo`) (voir `min$`).

$sp > s$	1
$sp = s$	0
$sp < s$	-1

La fonction `cmp`(*s*, *sp*) est donc l'analogue de `sgn`($sp - s$) pour des réels *s* et *sp*.

La fonction `cmp1` compare les chaînes suivant la relation d'ordre des dictionnaires. Les signes auxiliaires et la distinction entre majuscule et minuscule n'interviennent qu'en dernier ressort. Elle équivaut à la fonction `cmp1_bis` suivante :

```
cmp1_bis:function(char x,y)
         value=cmp(upper1$(x),upper1$(y))
         ift value return
         value=cmp(x,y)
         return
```

Exemple

```
char s,sp
s="été"
sp="hiver"
print cmp(s,sp);cmp1(s,sp)
```

Sortie (20 ms)

```
 -1  1
```

LEN(exprchaîne)

V_fonction Nombre de caractères

La fonction `len` renvoie la longueur de la chaîne en octets.

Exemple

```
print len("12345" & chr$(0,7))
```

Sortie (10 ms)

```
 12
```

SPACE$(a)

SPC(a)

C_fonctions Chaîne de a espaces

a

entier*32 $a \geq 0$

En Basic 1000d les deux fonctions `space$` et `spc` sont identiques. Elles sont équivalentes à `chr$`(32, a), `string$`(a, 32) ou `string$`(a, “ ”).

Exemple

Création d'une chaîne de 100000 espaces.

```
print len(spc(100000))
```

Sortie (385 ms)

```
 100000
```

STRING$(n, k)

STRING$(n, c)

CHR$(k [, n])

CHRP$(k [, n])

C_fonctions Répète n fois

n

entier*32 ≥ 0 (défaut $n = 1$)

k

entier $k \in [0, 255]$

c

exprchaîne

La forme `string$`(n, k) ou `chr$`(k, n) renvoie une chaîne de caractères formée de n fois le caractère de code ASCII k. La forme `string$`(n, c) renvoie une chaîne de caractères formée de n fois la chaîne c. Si $n = 0$, ces fonctions renvoient la chaîne vide.

Exemple

```
print string$(10,48)
print string$(5," *")
```

Sortie (40 ms)

```
0000000000
 * * * * *
```

La fonction `chrp$`(k) permet d'afficher le caractère Atari de code k même lorsque k est un code de contrôle. Elle utilise le fait que dans la commande `print`, le code $19 indique que le caractère suivant ne doit pas être considéré comme code de contrôle, mais sorti tel quel. Si $k \geq 32$, `chr$` et `chrp$` sont identiques. Si $k < 32$, `chrp$`(k, n) est identique à `string$`(n, `chr$`($19) & `chr$`($k$)).

Exemple

Comparer

```
print chr$(13,4);
```

qui effectue 4 retours chariots et

```
print chrp$(13,4);
```

qui écrit 4 fois le symbole Atari de code 13.

ASC(exprchaîne)

V_fonction code ASCII

La fonction `asc` renvoie le code ASCII du premier caractère de la chaîne, ou 0 si la chaîne est vide.

Exemple

Pour une chaîne à un caractère, `asc` est l'inverse de `chr$`.

```
print asc("Ab")
print asc(chr$(123))
```

Sortie (30 ms)

```
65
123
```

Conversions avec chaînes

STR$(p, {, [/] xi})

C_fonction Conversion expr → chaîne

xi

littéral

p

expr

La forme `str$`(p) effectue la conversion de p en chaîne alphanumérique (forme qui peut s'écrire). C'est l'inverse de la fonction `val`. On peut souvent écrire p au lieu de `str$`(p) parce que lorsque le Basic 1000d attend une chaîne, p est alors automatiquement converti en chaîne. Par exemple :

```
c$=tan(1/2)
```

équivaut à :

```
c$=str$(tan(1/2))
```

La forme `str$`(p, [/]x, [/]y) permet d'ordonner les expressions suivant les littéraux x et y (forme dite récursive). Elle n'est utilisable que si p est une somme de termes de la forme $A_{i,j}x^iy^j$ où les exposants i et j peuvent être négatifs et où les $A_{i,j}$ ne dépendent pas de x et y.

C'est justement cette somme que renvoie `str`, en ordonnant les termes suivant les puissances décroissantes (ou croissantes si l'option / est donnée devant le littéral).

Exemple

Le premier `print` équivaut à `print W`.

```
W=a*X+a^2/X+sum(I=-3,1,2 of X^I/I)
print str$(W)
print str$(W,X)
print str$(W,/a)
```

Sortie (400 ms)

```
 1/3* [X]^-3* [3*a^2*X^2 +3*a*X^4 +3*X^4 -3*X^2 -1]
(   [a +1])*X+(   [a^2 -1])*X^-1+( -1/3)*X^-3
(  1/3* [X]^-3* [3*X^4 -3*X^2 -1])+(   [X])*a+(   [X]^-1)*a^2
```

VAL(exprchaîne)

V_fonction Conversion chaîne → expr

La fonction `val` renvoie la valeur algébrique d'une exprchaîne (codage alphanumérique). C'est l'inverse de la fonction `str$`.

Exemple

```
print val("-" & "x^12" & "/7")
```

Sortie (35 ms)

```
-1/7*x^12
```

BIN$(p)

OCT$(p)

HEX$(p)

C_fonctions Conversion expr $p \rightarrow$ base 2, 8 et 16

p

expr

La fonction `bin$` (resp `oct$`, `hex$`) renvoie une chaîne contenant l'expr p écrite en binaire (resp octal, hexadécimal). L'expr p est décodée avec la valeur courante de la base.

Exemple

Affiche 10^{15} en binaire, octal et hexadécimal.

```
print bin$(10^15)
print oct$(10^15)
print hex$(10^15)
```

Sortie (125 ms)

```
11100011010111111010100100110001101000000000000000
34327724461500000
38D7EA4C68000
```

Exemple

Donnons ici la généralisation, `base$`(p, b), de ces fonctions à une base b quelconque. Ainsi `base$`(p, `sixteen`) est identique à `hex$`(p). L'exemple convertit un nombre de base 10 en base 36.

```
    print base$(101560761707988,36)
    stop
base$:function$
    push base,@1
    base @2
    value=str$(pop)
    base pop
    return
```

Sortie (50 ms)

```
1000DBASIC
```

MKX$(x)

C_fonction Conversion expr $x \rightarrow c$

CVX(c)

V_fonction Conversion $c \rightarrow x$

x

expr

c

exprchaîne, codage mémoire d'une expr

Les fonctions `mkx$` et `cvx` sont inverses l'une de l'autre. Les noms de ces fonctions viennent de convert (CV), make (MK) et expression (X). La fonction `mkx$`(x) renvoie le codage mémoire de x.

Exemple

Les représentations mémoires de x et c sont identiques.

```
x=(z+1)/(z-1)
char c
c=mkx$(x)
print x
print cvx(c)
```

Sortie (75 ms)

```
[z -1]^-1* [z +1]
[z -1]^-1* [z +1]
```

Exemple

```
save$ "trente.exp",mkx$((1+x)^30)
W=cvx(load$("trente.exp"))
```

crée et relit un fichier plus court que si on utilise :

```
save$ "trente.exp",(1+x)^30
W=val(load$("trente.exp"))
```

mais, le nom du littéral x est perdu.

MKZ$(p, k)

C_fonction Conversion entier $p \to c$

CVZ(c)

V_fonction Conversion $c \to p$

k

entier dans [0,4096[

p

entier

c

exprchaîne

La fonction `mkz$`(p, k) renvoie une chaîne de k octets représentant `modr`(p, 256^k). Cette représentation est la généralisation du codage machine des entiers signés sur 1, 2 ou 4 octets à k octets. Elle diffère de la représentation interne des nombres du Basic 1000d. Comparer les codages (donnés par octets) des deux

nombres 0 et 2^{13} :

p	mkz$($p$, 2)	codage interne
0	0, 0	$40, 0
$2000	$20, 0	2, $20, 0

Si c est la chaîne des k octets a_{k-1}, a_{k-2}, ..., a_1, a_0, la fonction cvz(c) renvoie l'entier relatif $p \in [-2^{8k-1}, 2^{8k-1}[$ tel que $p = \sum_i a_i 2^{8i} \pmod{2^{8k}}$. Dans :

```
p=cvz(c)
cp=mkz$(p,len(c))
```

les chaînes c et cp sont identiques.

Le z dans les mots clefs cvz et mkz$ rappelle la notation mathématique de l'ensemble des entiers relatifs. Parmi les fonctions de conversions, les fonctions mkz$ et cvz sont celles qui ont le plus d'applications, en particulier en cryptographie et en transmission.

Exemple

Pour transmettre la chaîne de 128 octets, contenue dans c$, on rajoute un mot de vérification de deux octets, check. Les 128 octets sont considérés comme un nombre n très grand, et check s'obtient comme le reste signé modulo le plus grand nombre premier codé sur 16 bits (65521) de $n2^{16}$. L'appel xbios($F) fixe la configuration du port série RS232, avant la transmission des données. Le protocole XMODEM utilise une méthode analogue (dite de redondance cyclique).

```
c$=string$(32,"abcd")
n=cvz(c$)
check=mods(n*2^16,65521)
ift xbios(15,0,0,$86,-1,-1,-1)
open "o",#1,"aux:"
print #1,c$;mkz$(check,2);
```

MKI$(ws)

MKL$(ls)

C_fonctions Conversion $ws \to cw$ ou $ls \to cl$

CVI(cw)

CVL(cl)

V_fonctions Conversion $cw \to ws$ ou $cl \to ls$

ws

entier*16

ls

entier*32

cw, cl

exprchaînes, `len`(*cw*)=2 et `len`(*cl*)=4

cas	entier	chaîne
`Cvi/Mki$`	$[-2^{15}, 2^{15}[$	2 octets
`Cvl/Mkl$`	$[-2^{31}, 2^{31}[$	4 octets

La fonction `mki$` (resp `mkl$`) renvoie une chaîne de 2 (resp 4) caractères représentant le codage machine d'un entier signé. `mki$`(*ws*) est identique à `mkz$`(*ws*, 2), lorsque *ws* est un entier*16. De même `mkl$`(*ls*) et `mkz$`(*ls*, 4) sont identiques, mis à part que *ls* est limité aux entiers*32 dans la première forme.

Les fonctions `cvi` et `cvl` sont inverses des fonctions `mki$` et `mkl$` et renvoient respectivement un entier*16 et un entier*32. Les lettres `i` et `l` dans les mots `cvi`, `cvl`, `mki$` et `mkl$` rappellent les mots Integer et Long.

Exemple

```
char c
c=mki$($6789)
print /h/ cvi(c)
c=mkl$($12345678)
print /h/ cvl(c)
```

Sortie (50 ms)

```
6789
12345678
```

MKS$(p)

MKD$(p)

C_fonctions Conversion réel $p \to$ format IEEE

CVS(s$)

CVD(d$)

V_fonctions Conversion format IEEE $\to p$

p

réel

s$

exprchaîne de 4 octets

d$

exprchaîne de 8 octets

Les fonctions `mks$` et `mkd$` convertissent un réel aux formats recommandés par l'IEEE pour les flottants simple ou double précision. Ces formats ne sont pas

utilisés par le Basic 1000d, mais les fonctions de conversions permettent la jonction avec des terminaux ou programmes utilisant ces formats. Le nombre 0 est codé par 32 ou 64 bits nuls. Le nombre $p = s2^m(1+x)$ où s est le signe, m un entier et $x \in [0, 1[$ est codé par :

Format 32 bits bits		Format 64 bits bits	
31	s	63	s
30–23	$m + 127$	62–52	$m + 1024$
22–0	`cint`$(x2^{23})$	51–0	`cint`$(x2^{52})$

Les C_fonctions `mks$` et `mkd$` renvoient une chaîne de 4 ou 8 octets contenant ces formats. Les fonctions `cvs` et `cvd` effectuent les conversions inverses.

Exemple

On écrit le plus petit réel strictement positif représenté dans ces formats, ainsi que le plus petit réel négatif. Le premier nombre est représenté par des zéros, sauf le bit 0, et le deuxième par tous les bits égaux à 1.

```
print "32 bits";cvs(mkz$(1,4));cvs(mkz$(-1,4))
print "64 bits";cvd(mkz$(1,8));cvd(mkz$(-1,8))
```

Sortie (385 ms)

```
32 bits  0.5877472455~ E-38 -0.6805646933~ E+39
64 bits  0.1112536929~ E-307 -0.3595386270~ E+309
```

Copie, tri, recherche et permutations

Les commandes de cette section, agissent sur N éléments d'un tableau. Les tableaux peuvent avoir un nombre quelconque d'indices, et être de types var, char ou index**size* avec $size = 8$, 16 ou 32. La donnée des éléments du tableau se fait par le premier élément, le nombre d'éléments N, et un pas p (un entier positif, nul ou négatif). Par exemple :

```
T(2),7,1
```

désigne les 7 éléments T(2), T(3), ..., T(8) et :

```
var TA(2,8)
TA(0,0),8,3
```

désigne les 8 éléments TA(0,0), TA(0,1), ..., TA(0,7) (le pas étant égal à 3, les éléments TA(1,i), TA(2,i) sont sautés). La forme :

```
TA(2,8),27,-1
```

désigne tous les 27 éléments du même tableau dans l'ordre inverse et :

```
TA(0,0),10,0
```
représente 10 fois l'élément TA(0,0).

COPY S(k...), N, p, D(j...), q

Commande Copie de N éléments de S vers D.

N

entier $N \geq 0$, nombre d'éléments à copier

S(k...)

premier élément à copier

p

pas entre les éléments à copier

D(j...)

premier élément du tableau destination

q

pas entre les éléments destination

Par exemple :
```
var D(10)
index S(100)
copy S(2),10,5,D(1),1
```
où la commande `copy` effectue la boucle suivante (mais plus vite) :
```
for i=1,10
  D(i)=S(2+(i-1)*5)
next i
```
La copie est faite un élément après l'autre, en suivant le même ordre que dans la boucle `for`. Cette particularité permet des effets intéressants lorsque les tableaux source et destination sont identiques. On peut ainsi effacer un élément et décaler d'un cran vers le début toutes les valeurs du tableau après l'élément. On peut au contraire décaler d'un cran vers la fin toutes les valeurs du tableau après un élément, ce qui permet d'insérer une nouvelle valeur dans le tableau. Ces possibilités sont mises en œuvre dans l'exemple suivant.

Exemple

Les dix éléments du tableau modifié par `copy` sont affichés après chaque modification.
```
index S(9),D(9)
for i=0,9
  S(i)=9-i
next i
copy S(0),10,1,D(0),1
print "copy       ";conc$(i=0,9 of D(i))
copy D(9),10,-1,S(0),1
print "inverse    ";conc$(i=0,9 of S(i))
```

```
        copy S(3),7,1,S(2),1
        print "ôte S(2)   ";conc$(i=0,9 of S(i))
        copy S(8),7,-1,S(9),-1
        S(2)=0
        print "insère S(2)";conc$(i=0,9 of S(i))
        copy S(0),8,1,S(2),1
        print "répète 0,1 ";conc$(i=0,9 of S(i))
```

Sortie (465 ms)

```
  copy         9  8  7  6  5  4  3  2  1  0
  inverse      0  1  2  3  4  5  6  7  8  9
  ôte S(2)     0  1  3  4  5  6  7  8  9  9
  insère S(2)  0  1  0  3  4  5  6  7  8  9
  répète 0,1   0  1  0  1  0  1  0  1  0  1
```

Remarquer que la commande :

```
        copy S(9),10,-1,S(0),1
```

ne peut pas être utilisée pour l'inversion du tableau.

SORT S(k...), N, p, P(j...) [, D [, fcmp]]

Commande Tri de N éléments de S

N

entier $N \geq 1$, nombre d'éléments à trier

S(k...)

premier élément à trier

p

pas entre les éléments à trier

P(j...)

premier élément dans un tableau d'index*32

D

entier $D \in [0, N-1]$ (défaut $D = 0$), nombre d'éléments déjà triés

fcmp

fonction donnant la relation d'ordre

La commande `sort` peut trier des réels ou des chaînes. Elle ne modifie pas le tableau S, mais renvoie dans le tableau d'index*32 P, dans les N éléments à partir de P(j...), une permutation des entiers de 1 à N qui indique l'ordre croissant des éléments à trier.

Le réarrangement du tableau (souvent inutile, car P suffit) a été dissocié du tri et est possible par la commande `permute` décrite plus bas. La combinaison de `sort` et `permute` permet également de réarranger plusieurs tableaux suivant l'ordre d'un des tableaux.

Pour rajouter quelques éléments dans une table déjà ordonnée, la donnée de D, le nombre d'éléments déjà ordonnés, doit être accompagnée du tableau

P(j...) contenant la permutation des entiers de 1 à D indiquant l'ordre des D premiers éléments du tableau. Le tri est alors plus rapide qu'un tri total (avec $D = 0$).

Exemple

Tri d'un tableau S de 100 entiers aléatoires

Les nombres sont affichés dans l'ordre croissant.

```
var S(100)
index P(100)
for i=1,100
  S(i)=random(2^100)
next i
sort S(1),100,1,P(1)
for i=1,100
  print S(P(i))
next i
```

Sortie (7 s)

```
1232854174249763393673334520433
1257783496757281630891887857197
...
```

Dans le cas d'égalité, P préserve l'ordre initial. Pour l'exemple ci-dessus, si S(P(i))=S(P($i+1$)), alors P(i)<P($i+1$).

On peut dans le cas où S est un tableau de variables (S index est exclu et provoque une erreur Instruction Illégale) effectuer un tri avec une relation d'ordre quelconque, calculée par la fonction `fcmp` indiquée dans le dernier argument. Il faut alors donner D, même s'il est nul. L'expression `fcmp`(x, y) doit être un entier*32 qui spécifie la relation d'ordre entre les expr (resp exprchaînes) x et y si S est de type var (resp char), selon la table suivante :

	`fcmp`(x, y)
$y > x$	> 0
$y = x$	$= 0$
$y < x$	< 0

Le tri ordinaire, avec `fcmp` omis, est donc équivalent à l'utilisation de `cmp` pour `fcmp` si S est de type char, et à celle de la fonction `sgn1` suivante si S est de type var :

```
sgn1:function(x,y)
     value=sgn(y-x)
     return
```

Exemple

Le programme range le tableau S de type char suivant trois relations d'ordre. Le premier rangement est effectué suivant la relation d'ordre implicite

cmp qui dépend de distingo. Le deuxième est effectué suivant la relation d'ordre de la fonction interne cmp1, qui convient parfaitement pour classer les mots français. Le troisième rangement est un exemple de relation d'ordre spécifiée par une fonction externe en Basic. Les mots sont rangés suivant leur longueur, puis suivant cmp1.

```
        char S(14)
        index P(14)
        for i=1,14
          read S(i)
        next i
        data été,êtes,étés,Etéocle,odeur,de,sainteté
        data Avoir,l,œil,sur,Œdipe,offre,et cætera
        sort S(1),14,1,P(1)
        print "CMP ";conc$(i=1,14 of ffff&S(P(i)))
        sort S(1),14,1,P(1),0,cmp1
        print "CMP1";conc$(i=1,14 of ffff&S(P(i)))
        sort S(1),14,1,P(1),0,CMPL
        print "CMPL";conc$(i=1,14 of ffff&S(P(i)))
        stop
    CMPL:function(char x,y)
        value=sgn(len(upper1$(y))-len(upper1$(x)))
        ift value return
        value=cmp1(x,y)
        return
```

Sortie (1665 ms)

```
 CMP    Avoir    Etéocle    de    et cætera    l    odeur    offre
  sainteté    sur    été    étés    êtes    œil    Œdipe
 CMP1    Avoir    de    et cætera    été    Etéocle    étés    êtes
  l    odeur    Œdipe    œil    offre    sainteté    sur
 CMPL    l    de    été    sur    étés    êtes    œil    Avoir    odeur
    offre    Œdipe    Etéocle    sainteté    et cætera
```

PERMUTE S(k...), N, p, P(j...)

Commande Permute N éléments de S

Les arguments de la commande permute sont identiques à ceux de la commande sort, mais les N éléments à partir de P(j...), doivent maintenant être une permutation des entiers de 1 à N qui indique l'ordre de réarrangement à effectuer.

Une commande permute effectuée après une commande sort permet de ranger un tableau dans l'ordre croissant.

Exemple

En utilisant mid$, le tableau S est rempli avec les caractères de “Basic 1000d”, puis ces caractères sont triés par sort et ordonnés par permute. Le tableau auxiliaire P contient une permutation des nombres 1 à 11.

```
        char S(10)
```

```
index P(10)
for i=0,10
  S(i)=mid$("Basic 1000d",i+1,1)
next i
sort S(0),11,1,P(0)
permute S(0),11,1,P(0)
print conc$(i=0,10 of S(i))
```

Sortie (150 ms)

```
0001Bacdis
```

SEARCH(S(k...), N, p, P(j...) [, fcmp])

V_fonction Recherche une valeur dans un tableau trié

Un des avantages d'un tableau trié est de permettre de retrouver rapidement une valeur donnée dans ce tableau. La fonction `search` permet d'effectuer cette recherche en un temps proportionnel à `intlg`(N). Les arguments de `search` sont les mêmes que ceux qui ont permis de trier le tableau de $N-1$ éléments S(k...), mis à part que le nombre d'éléments était $N-1$ au lieu de N maintenant, et que le tableau S contient un élément de plus, la valeur à rechercher. Cet élément est le N-ième élément dans la suite S(k...), ... de pas p. Par exemple dans :

```
k=SEARCH(S(1),100,1,P(1))
```

S(100) est recherché dans la table des 99 éléments S(1) à S(99). Le tableau P décrivant l'ordre de la table a pu être rempli par :

```
SORT S(1),99,1,P(1)
```

Pour utiliser `search`, il faut donc prévoir un élément supplémentaire dans le tableau S.

La fonction `search` renvoie l'entier $k \in [0, N]$ qui indique la position où s'insère l'élément.

Exemple

Les 99 mots S(1) à S(99) sont triés, puis on détermine où s'insère le mot S(100) qui est aléatoire la première fois, entré au clavier ensuite. La valeur k renvoyée par `search` est telle que $\mathrm{S}(\mathrm{P}(k)) \leq \mathrm{S}(100) < \mathrm{S}(\mathrm{P}(k+1))$ si on fait la convention que S(P(0)) désigne $-\infty$ et S(P(100)) désigne $+\infty$. On remarquera également qu'il faut appeler `sort`, mais pas `permute`, avant `search`.

```
char S(100)
index P(100)
for i=1,100
  S(i)=conc$(j=0,random(7) of chr$($61+random(26)))
next i
sort S(1),99,1,P(1)
print "Dans la liste : ";conc$(i=1,99 of S(P(i))ƒ)
do
  k=search(S(1),100,1,P(1))
  ift k print S(100)," suit    ",S(P(k))
  ift k<>99 print S(100)," précède ",S(P(k+1))
```

```
    input "entrer un nom",S(100)
  loop
```

Exemple

Les 10 mots aléatoires de deux lettres S(1) à S(10) sont classés suivant l'ordre alphabétique inverse spécifié par la fonction cmpm. La fonction `search`, utilisée avec cette relation d'ordre permet de déterminer où se range le mot S(11).

```
          char S(11)
          index P(10)
          randomize 1
          for i=1,11
            S(i)=chr$($41+random(26))&chr$($41+random(26))
          next i
          sort S(1),10,1,P(1),0,cmpm
          print conc$(i=1 to 10 of S(P(i))ff)
          k=search(S(1),11,1,P(1),cmpm)
          print S(11);" suit ";S(P(k))
          print S(11);" précède ";S(P(k+1))
          stop
     cmpm:function(char x,y)
          value=cmp(mirror$(x),mirror$(y))
          return
```

Sortie (725 ms)

```
RG  ZO  DP  GP  DU  LU  NW  VW  WW  SX
KS suit GP
KS précède DU
```

Exemple

Il est montré comment rajouter un nouveau mot dans une table (dico) ordonnée, seulement si ce mot n'est pas déjà dans la table. La fonction `search` permet de déterminer si le mot entré est connu. Sinon, le nombre de mots N du dictionnaire est augmenté, et le tableau dico est trié partiellement seulement sur le nouveau mot, en indiquant par le dernier argument de `sort` que les $N-1$ premiers mots sont déjà triés.

```
  char dico(100)
  index P(100),i
  N=0
  do
    input "Entrer un mot",dico(N+1)
    i=search(dico(1),N+1,1,P(1))
    if dico(P(i))=dico(N+1)
      print "Mot déjà inscrit"
    else
      N=N+1
      sort dico(1),N,1,P(1),N-1
```

```
        endif
        print conc$(i=1,N of "   "&dico(P(i)))
      loop
```

NEXTPERM(N, P(j. . .) [, a])

V_fonction Génère une permutation

N

entier ≥ 1

P(j. . .)

premier élément dans un tableau d'index*32

a

entier*32, indique l'option (défaut $a = 1$)

La fonction `nextperm` remplit les N éléments consécutifs du tableau P, à partir de P(j. . .), avec une permutation des entiers de 1 à N, et prend pour valeur ± 1 qui donne la parité de la permutation.

Si $a = 0$, la permutation renvoyée est 1, 2, 3, . . . , N; c'est la première permutation suivant l'ordre lexicographique.

Si $a > 0$, le tableau P doit contenir en entrée une permutation de 1, 2, . . . , N. La fonction renvoie alors la permutation suivante des mêmes nombres, pour l'ordre lexicographique. Dans le cas où la permutation en entrée dans P était la dernière (N, $N-1$, . . . , 3, 2, 1), la fonction `nextperm` ne modifie pas P et prend la valeur 0.

Si $a < 0$, la fonction renvoie une permutation aléatoire de 1, . . . , N.

La fonction `nextperm` permet de réaliser très simplement une boucle sur les $N!$ permutations de 1, . . . , N. En pratique, on ne pourra guère réaliser de boucles complètes que pour $N < 10$ (la boucle vide sur les 9! permutations de 1, . . . , 9 nécessite 2000 s).

Exemple

Boucle sur les permutations de 1, 2, 3.

```
print "permutation   parité"
index P(3)
k=nextperm(3,P(1),0)
while k
  print conc$(i=1,3 of P(i));"      ";k
  k=nextperm(3,P(1))
wend
```

Sortie (295 ms)

```
 permutation   parité
   1  2  3        1
   1  3  2       -1
   2  1  3       -1
   2  3  1        1
```

```
3  1  2        1
3  2  1       -1
```

Exemple

Le programme suivant distribue au hasard les 52 cartes entre quatre joueurs. La carte numéro i ($0 \leq i < 52$) est la P(i)-ième carte distribuée, où P(i) est une permutation aléatoire de 1, ..., 52. Le premier joueur reçoit les 13 premières cartes, le deuxième joueur les 13 suivantes, etc.

```
index P(51),J
char D(3),N(12)
N(0)="A"
N(1)="R"
N(2)="D"
N(3)="V"
FOR I=4,12
  N(I)=justl$(14-I)
next I
DO
  J=0
  cls
  IFT NEXTPERM(52,P(0),-1)
  FORC C IN ("P","C","K","T")
    FOR I=0,3
      D(I)=C&"- "
    NEXT
    FOR J=J,J+12
      CADD D(DIVR(P(J)-1,13)),f&N(MODR(J,13))
    next J
    PRINT TAB(15),D(0)
    PUSH CURSL
    CURSL CURSL+4
    print D(1);TAB(30),D(2)
    CURSL CURSL+4
    PRINT TAB(15),D(3)
    CURSL POP
  nextC
  CURSL 20
  print "Pour une nouvelle donne presser une touche"
  ift keyget
loop
```

Exemple (monochrome)

Fait disparaître l'écran. L'écran est divisé en 80 cases qui sont effacées en désordre. `nextperm` suivi de `exg` permet de créer une permutation aléatoire des nombres de 0 à 79 qui fixe l'ordre d'effacement de ces cases. La fonction

`inp`(2), à la différence de `keyget`, attend l'appui sur une touche sans montrer le curseur ou la souris.

```
index P(79)
ift nextperm(79,P(1),-1)
exg P(0),P(random(80))
cursh 0
cls
hidecm
for i=0,79
  origin modr(P(i),8)*80,divr(P(i),8)*40
  pbox 0,0,80,40
next
ift inp(2)
cls
```

Exercice Max Detprm

Déterminer une permutation des éléments de la matrice :

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

telle que la valeur absolue du déterminant de la matrice soit le plus grand nombre premier possible.

Opérations dans les variables

Dans cette section, nous décrivons des commandes qui modifient directement la valeur d'une variable, comme par exemple :

```
vmul x,5
```

qui multiplie la variable `x` par 5. Cette forme présente l'avantage d'être plus rapide et plus économe en mémoire que la commande :

```
x=x*5
```

EXG a, b

Commande Echange

a, b

nomi du même type var, char, index*32, index*16 ou index*8

La commande `exg` échange les valeurs de `a` et `b`.

Exemple

Les numéros des variables donnés par `varnum` ne sont pas modifiés, seuls les contenus sont échangés.

```
var T(1)
T(0)=1
T(1)=2
print T(0);T(1);varnum(T(0));varnum(T(1))
exg T(0),T(1)
print T(0);T(1);varnum(T(0));varnum(T(1))
```

Sortie (85 ms)

```
 1  2  537  536
 2  1  537  536
```

CADD c$, virchaîne

Commande Concaténation c$=c$ & virchaîne

VADD v, expr

Commande Addition v=v + expr

VSUB v, expr

Commande Soustraction v=v − expr

VMUL v, expr

Commande Multiplication v=v ∗ expr

VDIV v, expr

Commande Division v=v / expr

VDIVE v, expr

Commande Division exacte v= `dive`(v, expr)

c$

nomi de type char

v

nomi de type var (index non permis)

La commande `cadd` rallonge c$ à droite.

Exemple

```
c$="A"
cadd c$,"b","c"
print "c$=";c$
```

Sortie (20 ms)

```
c$=Abc
```

Les commandes `vadd`, `vsub`, `vmul`, `vdiv` et `vdive` effectuent des opérations élémentaires de la façon la moins élaborée possible. A la différence des assignations, les conditions `cond` sont ignorées et il n'y a pas conversion suivant la variable d'état `develop` (ou `factor`).

La commande `vdive` effectue la division exacte lorsque `v` et expr sont des polynômes. Sinon (formes factorisées ou flottantes), `vdive` effectue la division ordinaire v=v/expr.

Exemple

Dans `vmul` la condition $x^3 = -2$ a été ignorée. La valeur `v` n'est donc pas la plus simple possible.

```
cond x^3+2
w=(1+x)^2
w=w*w
print "w=";w
v=(1+x)^2
vmul v,v
print "v=";v
```

Sortie (130 ms)

```
w=  6*x^2 +2*x -7
v=  x^4 +4*x^3 +6*x^2 +4*x +1
```

Exemple

Calcule l'approximation de e donnée par $1+1+1/2!+1/3!+\cdots+1/12!$. Les lignes 4 et 5 du programme sont équivalentes à :

```
v=v/i
w=w+v
```

tout en étant très légèrement plus rapides.

```
w=1~
v=1~
for i=1,12
  vdiv v,i
  vadd w,v
next
print w
```

Sortie (130 ms)

```
0.2718281828~ E+1
```

Pile utilisateur

On dispose d'une pile pouvant contenir jusqu'à `s_var` variables (`s_var` est modifiable). Cette pile utilisateur est uniquement utilisée par les six commandes et fonctions ci-dessous.

PUSH expr {, expr}

PUSH$ exprchaîne { , exprchaîne }

Commandes Poussent des valeurs dans la pile

POP

V_fonction

POP$

C_fonction

STACK(n)

STACK$(n)

Accès à la pile (lecture et écriture)

n

entier*16

Les expressions poussées dans la pile par les commandes `push` et `push$` peuvent être retrouvées par les fonctions `pop` et `pop$` qui ôtent une valeur de la pile et renvoient cette valeur. La fonction `pop` vérifie que la valeur est une expr. On peut profiter de la pile pour effectuer des conversions :

```
push (a+b)^7
c$=pop$
```

est équivalent à :

```
c$=mkx$((a+b)^7)
```

Exemple

La pile est remplie dans l'ordre avec 5, 2, 15 et "x=". Elle est vidée dans l'ordre inverse (les expressions sont toujours décodées de gauche à droite), ce qui revient à calculer 15^2/5.

```
push 5
push 2,15
push$ "x="
print pop$;pop^pop/pop
```

Sortie (25 ms)

```
 x=  45
```

Exemple

La pile permet d'effectuer une permutation de variables.

```
x=1
y=2
z=3
push x,y,z
y=pop
x=pop
z=pop
print x;y;z
```

Sortie (25 ms)

```
 2  3  1
```

L'argument n dans `stack` et `stack$` doit être compris entre 0 et $p-1$, où p désigne le nombre d'éléments dans la pile. L'argument 0 correspond au dernier élément poussé dans la pile, et l'argument $p-1$ au plus ancien élément poussé. On peut utiliser `stack`(n) et `stack$`(n) comme fonctions pour lire une valeur dans la pile, sans modifier la dimension de la pile. On peut aussi utiliser les commandes :

```
stack$(n)=virchaîne
stack(n)=expr
```

pour modifier une valeur de la pile.

Exemple

On transforme avec `stack` des chaînes en expr dans la pile. Noter que dans `stack`(n) = expr, lorsque n et/ou expr modifient la hauteur de la pile, l'élément modifié par la commande est l'élément numéro n, après les rectifications de hauteur de la pile.

```
push$ "  XY","A"
stack$(0)=stack$(0)&"BC"
print stack$(0);stack$(1);
stack(1)=5
stack(0)=7
push 0
stack(pop)=pop*stack(0)
print pop
```

Sortie (50 ms)

```
ABC  XY  35
```

Exemples d'application de la pile

L'intérêt principal de la pile est de permettre de conserver des valeurs sans créer des variables.

Exemple

Le programme conserve dans la pile les valeurs de x, `base` et `showm`. Après modification de ces variables, les valeurs initiales sont restaurées. Noter l'ordre inverse pour relire la pile.

```
x=145
push x,base,showm
hidem
base 7
x=21
print /d/x
showm pop
base pop
x=pop
```

Sortie (25 ms)

```
15
```

Exemple

Voici un programme créant la fonction `dete` qui renvoie la valeur d'un déterminant dont on donne tous les éléments comme argument de la fonction. La fonction s'appelle elle-même dans le calcul de :

$$\begin{vmatrix} \begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix} & \dfrac{x+1}{x-1} \\ 1-x^2 & 5 \end{vmatrix}$$

```
          print dete(dete(1,2,3,4),(x+1)/(x-1),1-x^2,5)
          stop
     dete:function
          if root(@0,2)
            local index dete_detn
            for dete_detn=1,@0
              push @dete_detn
            next dete_detn
            dete_detn=root(@0,2)
            value=det(dete_dete,dete_detn)
            return
          else
            print "*ERREUR* NB D'ARGUMENTS DE DETE"
            stop
          endif
dete_dete:function
          value=pop
          return
```

Sortie (190 ms)

```
x^2 +2*x -9
```

Une autre application intéressante de la pile est de permettre dans les sous-programmes, des calculs sans définir aucune nouvelle variable. Cela évite les erreurs qui se produisent lorsqu'on utilise le passage d'arguments par nom, quand ce nom contient, par malheur, le nom d'une variable locale.

Un exemple de ce type est discuté avec la fonction `hunt` présentée à propos de la commande `call`. L'utilisation de la pile y permet le décodage d'un argument de la forme `a to b`, sans nouvelles variables.

Peek et Poke

PEEKB(a)
PEEKBS(a)
PEEKW(ap)
PEEKWS(ap)
PEEKL(ap)
PEEKLS(ap)

V_fonctions Peek sur 1, 2 ou 4 octets.

POKEB a {, xi}
POKEBS a {, xi}
POKEW ap {, xi}
POKEWS ap {,xi}
POKEL ap {, xi}
POKELS ap {, xi}

Commandes Poke sur 1, 2 ou 4 octets

a

adresse

ap

adresse paire

xi

entier

Les fonctions `peekbs`, `peekws` et `peekls` se terminant par `s` renvoient un entier signé. Les fonctions `peekb`, `peekw` et `peekl` renvoient un entier positif ou nul. Les suffixes `b` (Byte =octet), `w` (Word =mot 2 octets) et `l` (Long= mot 4 octets) indiquent la longueur lue.

Les commandes `pokeb`, `pokebs`, `pokew`, `pokews`, `pokel` et `pokels`, correspondent aux fonctions PEEK avec même suffixes. Les valeurs permises de xi dépendent des suffixes S, B, W et L, comme indiqué dans la table suivante

	B	W	L
	$[\ 0,256[$	$[\ 0,65536[$	$[\ 0,4294967296[$
S	$[-128,128[$	$[-32768,32768[$	$[-2147483648,2147483648[$

Exemple

La valeur initiale de `limit` est conservée dans la pile pour pouvoir être restaurée en fin du programme. Elle est ensuite diminuée pour créer une zone

de 100 octets où on peut poker en toute liberté. `pokeb` écrit les 4 octets 1, 2, 3 et 4 à partir de l'adresse `limit`. `pokew` écrit le mot 5 en `limit` + 4 et le mot 6 en `limit` + 6. `pokel` écrit les mots longs 7 et 8 en `limit` + 8 et `limit` + 12. Suivent ensuite des écritures d'entiers signés. Ces écritures sont vérifiées à l'aide de `peekb`. On montre ensuite la différence entre les PEEK signés et non signés.

```
push limit
limit limit-100
pokeb limit,1,2,3,4
pokew limit+4,5,6
pokel limit+8,7,8
pokebs limit+16,-1,-128
pokews limit+18,-2
pokels limit+20,-4
print conc$(i=0,24 of peekb(limit+i))
print/h/ peekbs(limit+§16);peekb(limit+§16)
print/h/ peekws(limit+§18);peekw(limit+§18)
print/h/ peekls(limit+§20);peekl(limit+§20)
limit pop
```

Sortie (300 ms)

```
 1  2  3  4  0  5  0  6  0  0  0  7  0  0  0  8  255  128  255  254
255  255  255  252  0
-1  0FF
-2  0FFFE
-4  0FFFFFFFC
```

PEEK$(a, l)

PEEKZ$(a)

C_fonctions Peek chaîne

POKECB [a] a, virchaîne

POKECW [a] ap, virchaîne

Commandes Poke chaîne

a

adresse

l

entier*32

ap

adresse paire

a

C'est le symbole [a] L

La fonction `peek$` renvoie la chaîne de *l* octets commençant à l'adresse *a*. La fonction `peekz$` renvoie la chaîne formée des octets à partir de l'adresse *a* jusqu'au premier octet nul rencontré. Cette chaîne ne contient aucun octet nul.

Les commandes `pokecb` et `pokecw` implantent virchaîne dans la mémoire à partir de l'adresse *a* ou *ap* à raison d'un caractère par octet pour `pokecb` ou d'un caractère par mot pour `pokecw`, l'octet fort du mot étant mis à zéro. On implante également un octet ou mot nul après la chaîne sauf si l'option a ([a] L) est mise.

Exemple

Les 100 octets à partir de l'adresse *a* représentent les valeurs de `t`(i) pour i allant de 1 à 100. Le premier `pokecb` remplit tous ces octets avec −1. Cette méthode permet d'initialiser un grand tableau d'index beaucoup plus rapidement que par une boucle. L'option [a] L est ici indispensable, sinon un octet nul serait implanté en dehors de la zone attribuée au tableau `t`, ce qui peut détruire des données et conduire à une erreur Fatale. Le deuxième `pokecb` implante une chaîne, avec un zéro terminal. L'effet de ces `pokecb` est examiné par simple lecture de `t`(i). `peekz$` est utilisé pour relire la chaîne implantée. Ensuite, une combinaison de `pokecb` et `peek$` permet de recopier les 100 octets à partir de l'adresse 0 dans le tableau `t`.

```
index*8 t(100)
a=ptr(t(1))
pokecb a a,chr$($ff,100)
pokecb a,"ABCD"
print conc$(i=1,10 of t(i))
print peekz$(a)
pokecb a a,peek$(0,100)
print/h/ conc$(i=1,10 of t(i))
```

Sortie (205 ms)

```
   65  66  67  68  0 -1 -1 -1 -1 -1
 ABCD
   60  1E  1  0  0 -4  0  20  0  1 -6C -60  0  1 -6C -60
```

Décodage et Exécution

Les commandes et fonctions de ce paragraphe peuvent, entre autres, faciliter la réalisation de bibliothèques de programmes et d'émulateurs.

XQT virchaîne

Commande Exécution d'instructions Basic

virchaîne doit contenir une ou plusieurs instructions Basic séparées par `chr$`(0). La commande `xqt` exécute ces instructions.

Exemple

Utiliser le débogueur pour voir que :

```
xqt "print 1" & chr$(0) &"print 2"
```

exécute les instructions :

```
print 1
print 2
```

Exemple goto calculé

La sortie obtenue correspond à l'exécution de `goto L2` dans la commande `xqt`.

```
   char LBL(1)
   LBL(0)="L1"
   LBL(1)="L2"
   xqt "goto "& LBL(random(2))
L1:print 1
   stop
L2:print 2
   stop
```

Sortie (35 ms)

```
2
```

Exemple

L'appel de `vdi(6)` pour 60 points trace une spirale en 9 s.

```
hidecm
xqt "vdi 6,60" & conc$(i=1,60 of "," & justl$(cint(200+
 2*i*cos(i))& "," &justl$(cint(200+2*i*sin(i))))
```

Les instructions dans virchaîne sont cachées et ne peuvent définir des labels. Les structures (`if`, `select case`, `forv`, ...) doivent être complètes. Un programme comme

```
xqt "do"
print 1
loop
```

est accepté, mais le retour après `loop` n'est pas garanti. Il faut écrire :

```
xqt "do"æ,"print 1"æ,"loop"
```

Les remplacements de `@k` et `@@` sont bien sûr effectués sur le texte `xqt virchaîne`. De plus, si virchaîne est formé de plusieurs instructions, les remplacements de `@k` et `@@` sont effectués une deuxième fois pour ces instructions exceptée la première.

Exemple

La procédure `ex` écrit trois fois l'argument 1. Dans la troisième ligne de virchaîne il y a eu au total deux remplacements, le premier avant le décodage de `xqt`, et le deuxième avant le décodage du `print`.

```
        ex 17
        stop
     ex:xqt "print @1;"æ,"print @1;"æ,"print @@1;"
        return
```

Sortie (45 ms)

```
17  17  17
```

INSTR(s, sp [, a])
INSTRK(s, sp [, a])
RINSTR(s, sp [, a])
RINSTRK(s, sp [, a])

V_fonctions Position de *sp* dans *s*

s, sp

exprchaîne

a

entier*32 $a > 0$

La fonction `instr` cherche la chaîne *sp* dans la chaîne *s* à partir du *a*-ième caractère ($a = 1$ par défaut). Si la chaîne est trouvée elle renvoie la position *p* du début de la chaîne *sp* dans *s* (un entier $p > 0$). Si la chaîne *sp* n'est pas dans *s* elle renvoie 0. La recherche distingue ou non les majuscules des minuscules suivant la valeur de `distingo` (ou `nodistingo`).

La fonction `instrk` diffère de `instr` par le fait que devant et après la chaîne *sp* dans *s* il doit y avoir un symbole non alphanumérique (ceux qui servent dans les noms du basic), ou une extrémité de *s*. `instrk` est particulièrement adapté à la recherche de mots clefs, lors de la construction d'émulateurs.

Les fonctions `rinstr` et `rinstrk` sont analogues à `instr` et `instrk` mais cherchent la chaîne *sp* à partir du *a*-ième caractère de *s* (par défaut $a =$`len`(*s*)) en sens inverse (en allant vers la gauche). S'il y a une ou plusieurs occurrences de *sp* dans *s*, `rinstr` renvoie la position *p* de l'occurrence de *p* maximum ($0 < p \leq a$). *p* est la position du premier caractère de *sp* dans *s*, comme pour `instr`. En cas d'absence, ces fonctions renvoient 0.

Exemple

Après `nodistingo` les minuscules et majuscules sont équivalentes.

```
c$="from afrom to ato"
nodistingo
print instr(c$,"FROM");instrk(c$,"FROM")
print instr(c$,"FROM",2);instrk(c$,"FROM",2)
print rinstr(c$,"TO");rinstrk(c$,"TO")
```

Sortie (65 ms)

```
1  1
7  0
16  12
```

DECODE(s, sp [, a])

V_fonction Décode la chaîne *sp*

s, sp

exprchaîne

a

entier $a > 0$ (défaut $a = 1$)

Si à partir du a-ième caractère de la chaîne s on trouve la chaîne *sp*, éventuellement précédée d'espaces (mais de rien d'autre), et que la chaîne *sp* soit suivie d'un (ou terminée par un) séparateur, la fonction `decode` renvoie la position p ($p \geq a$) du premier caractère situé après *sp*. Sinon `decode` renvoie 0. Les majuscules/minuscules sont traitées suivant la variable d'état `distingo` (ou `nodistingo`).

Exemple

```
c$="clef  x"
print decode(c$,"clef");decode(c$,"x")
```

Sortie (20 ms)

```
 5  0
```

DECODEX(s [, a [, V]])

DECODEXI(s [, a [, I]])

DECODEXC(s [, a [, C]])

DECODELBL(s [, a [, I]])

DECODELIT(s [, a [, I]])

DECODEV(s [, a [, I]])

DECODEC(s [, a [, I]])

DECODEI(s [, a [, I]])

V_fonctions Décodent un élément

s

exprchaîne

a

entier $a > 1$ (défaut $a = 1$)

V

nomi de type var

C

nomi de type char

I

nomi de type index*32

La table suivante indique les éléments décodés suivant la fonction :

fonction	élément décodé
`decodex`	expr
`decodexi`	entier*32
`decodexc`	exprchaîne
`decodelbl`	nom de label
`decodelit`	littéral
`decodev`	nomi de type var
`decodec`	nomi de type char
`decodei`	nomi de type index

Si l'élément est écrit dans la chaîne *s* à partir du *a*-ième caractère, éventuellement précédé d'espaces, mais de rien d'autre, la fonction renvoie la position dans *s* du premier caractère suivant l'élément. Sinon, la fonction renvoie 0.

Dans les cas où le décodage a réussi, et où `C`, `I` ou `V` est donné en troisième argument, des renseignements supplémentaires sont assignés à `C`, `I` ou `V`. Pour `decodex` (resp `decodexi`, `decodexc`) c'est la valeur de l'élément qui est assignée à `V` (resp `I`, `C`).

△ Pour `decodelbl`, l'adresse d'un mot long qui contient l'adresse du ":" après le label dans le programme est mise dans `I`, et pour `decodelit`, le numéro du littéral. Pour `decodev`, `decodec` et `decodei`, une adresse dans la description interne du mot clef est assignée à `I`.

Exemple

Le nom `test` est à la fois une expr, une exprchaîne (comme toute expr), un littéral (au sens généralisé, parce que sa valeur est réduite à un nom de type lit), et un nomi de type var. L'espace après `test` dans `c` est considéré comme faisant partie de l'expr, mais pas du nomi de var.

```
char c,cp
c="test ,"
test=lis
print decodex(c,1,w);w
print decodexi(c)
print decodexc(c,1,cp);cp
print decodelbl(c)
print decodelit(c,1,j);j
print decodev(c,1,j);j;ptrptr(test)
print decodei(c)
```

Sortie (150 ms)

```
6  lis
```

```
0
6  lis
0
6  1
5  388438  388434
0
```

Exemple

La fonction `decodex` décode la chaîne la plus longue qui peut être interprétée comme expr. C'est `v(3)=8` et pas seulement `v(3)`.

```
    char c,cp
    index v(7)
    c="a:v(3)=8"
  a:v(3)=8
    print decodex(c,3,w);w
    print decodelbl("a",1,i)f;peekz$(peekl(i)+1)
    print decodei(c,3)
```

Sortie (75 ms)

```
9 -1
2 v(3)=8
7
```

Informations Mémoire

FRE

FRE(virchaîne)

V_fonction Nombre d'octets disponibles pour le programme.

La fonction `fre` effectue préalablement un ramassage des poubelles (`pack`). Les deux formes renvoient le même résultat, mais virchaîne est effectivement calculé dans la deuxième forme.

Exemple

Affiche l'espace libre avant et après création d'une chaîne de 50000 octets.

```
print fre
c$=space$(50000)
print fre
```

Sortie (255 ms)

```
158082
108076
```

MLEN(p)

V_fonction Longueur en octets du codage mémoire de *p*

p

expr

Si `W` est une variable de type var :

```
mlen(W)=peekls(ptr(W)-4)
```

On a aussi :

```
mlen(p)=len(mkx$(p))
```

Exemple

La longueur occupée en mémoire par *p*, `mlen`(*p*), diffère de la longueur de la représentation en caractères de *p*, `len`(*p*). Noter que dans `len`(*p*), l'argument *p* est considéré comme une exprchaîne.

```
p=(3+x)^100
print mlen(p);len(p)
```

Sortie (3875 ms)

```
2380  5328
```

PTR(x)

V_fonction Adresse mémoire du contenu de x

x

nomi de type char, var ou index

Pour étudier le codage de x vous pouvez effectuer la commande Query du menu TOOLS, avec pour entrée `ptr`(x).

Exemple

Le codage mémoire de `c$` est examiné par `peekl` et `peekb`. Le mot long en `ptr`(c$) − 4 est la longueur de la chaîne. Le contenu de c$ est formé des 8 octets en `ptr`(c$).

```
c$="ABCD"æ &"abc"
index i
i=ptr(c$)
print/h/"L=";peekl(i-4);"  c$:";conc$(i=i,i+7 of peekb(
 i))
```

Sortie (80 ms)

```
L=  8  c$:  41  42  43  44  0  61  62  63
```

Pour les variables, `ptr`(x) n'est pas fixe.

Exemple

Le changement de la valeur de `U` modifie l'adresse de `W`.

```
U=2^1000
W=3^333
V=ptr(W)
U=0
print V;ptr(W)
```

Sortie (40 ms)

```
391978  391838
```

Exemple

Création d'un tampon de 100000 octets accessible par PEEK et POKE. Le tampon est une variable de type char de longueur 100000. Comme son adresse n'est pas fixe, il ne faut pas définir une variable

```
J=ptr(TAMPON)
```

et l'utiliser dans :

```
pokeb J+7,$28
```

qui peut poker en dehors de TAMPON et conduire à un plantage.

```
char TAMPON
TAMPON=chr$(0,100000)
pokeb ptr(TAMPON)+7, $28
print peekb(ptr(TAMPON)+7)
```

Sortie (470 ms)

```
40
```

Cependant, comme `ptr` est précédé d'un `pack` implicite, la valeur renvoyée reste valable tant qu'aucune modification de variable (y compris système comme `log`(2) ou `pi` après un changement de `precision`) n'est effectuée. Dans le cas des index, `ptr`(x) est une adresse fixe.

Exemple Tampon fixe

Le programme crée une zone fixe de 100000 octets, d'adresse J. Pour un autre exemple, voir la fonction `gemdos($36)`.

```
index*8 tamp(99999)
J=ptr(tamp(0))
```

NEXTCODE

NEXTLINE

V_fonctions Adresse et numéro de l'instruction suivante.
Les fonctions `nextcode` et `nextline` sont utilisables seulement dans les procédures B_TRACE et B_DEBUG. Si l'instruction suivante n'est pas dans la source, `nextline` renvoie 0.

Exemples

Voir B_TRACE et B_DEBUG

Temps, Heure et Date

PAUSE t

Commande Attend t millisecondes

t

entier*32

Exemple

La procédure `slowprint` affiche une chaîne avec une attente de `t` ms entre chaque caractère. Les caractères sont isolés par `mid$`, et l'attente est effectuée par `pause`.

```
          slowprint 100,"Une pause vaut quatre soupirs"
          stop
slowprint:procedure(index t, char c)
          local index i
          for i=1,len(c)
            print mid$(c,i,1);
            pause t
          next
          print
          return
```

Sortie (3070 ms)

```
 Une pause vaut quatre soupirs
```

TIMER

V_fonction Temps en secondes

MTIMER

V_fonction Temps en millisecondes

CLEAR TIMER

Commande Mise à zéro des compteurs `timer` et `mtimer`

Les fonctions `timer` et `mtimer` renvoient le temps écoulé depuis `clear timer` ou `clear`. La valeur `mtimer` est toujours un entier multiple de 5. La valeur renvoyée par `timer` est égale à `int`(`mtimer`/1000).

Exemple

Détermine le temps nécessaire pour factoriser $1 - x^{12}$.

```
          clear timer
          print formf(1-x^12)
          print timer;mtimer
```

Sortie (4920 ms)

```
  - [x^2 +1]* [x -1]* [x +1]* [x^4 -x^2 +1]* [x^2 -x +1]* [x^2 +x +1]
   4  4905
```

DATE$ [=] s

TIME$ [=] s

Variables d'état Date et Heure

SETTIME s [, s]

Commande Fixe la date et l'heure

s

exprchaîne

L'exprchaîne *s* doit avoir pour valeur "hh:mm:ss" ou "mm/nn/yyyy". En tant que C_fonctions, `time$` et `date$` renvoient ces valeurs. En tant que commandes, `time$` et `date$` sont synonymes et le signe "=" est facultatif. La commande `settime` permet également de fixer l'heure et la date.

Exemple

C'est la forme de *s* qui indique si c'est la date ou l'heure. Ici les commandes `time$` et `date$` sont utilisées à contre emploi. L'heure est définie à 2 secondes près, d'où la valeur étrange du premier affichage de `time$`. Dans `settime` des formes simplifiées de *s* sont utilisées. `push$` et `pop$` sont utilisées pour ne pas trop modifier l'heure et la date du système.

```
push$ date$,time$
time$ "12/31/1988"
date$="23:59:59"
print date$ƒ;time$
pause 4000
print date$ƒ;time$
settime "10:20","3/31/88"
print date$ƒ;time$
settime pop$,pop$
```

Sortie (4085 ms)

```
12/31/1988 23:59:58
01/01/1989 00:00:02
03/31/1988 10:20:00
```

TIME_Y y

TIME_MO mo

TIME_N n

TIME_H h

TIME_M m

TIME_S s

Variables d'état Date et Heure

TIME_D

V_Fonction Jour de la semaine

y

Entier dans [1980,2107] donnant l'année

mo

Entier dans [1,12] donnant le mois

n

Entier dans [1,31] donnant le jour

h

Entier dans [0,23] donnant l'heure

m

Entier dans [0,59] donnant les minutes

s

Entier dans [0,59] donnant les secondes

Les variables d'état `time_y`, `time_mo`, `time_n`, `time_h`, `time_m` et `time_s` permettent de lire ou fixer individuellement l'année, le mois, le numéro de la date, les heures, les minutes et les secondes respectivement. La fonction `time_d` renvoie un entier dans [0,6] (0 pour Dimanche, 1 pour Lundi, ...)

Exemple

Le programme demande la date et l'affiche. Noter l'instruction `time_n 1` qui permet d'éviter une sortie erreur (si la date est le 31 mars, on ne peut pas changer le mois en mois d'avril sans changer le numéro).

```
input "Entrer la date jj,mm,yyyy",j,m,y
time_n 1
time_y y
time_mo m
time_n j
char jour(6),mois(12)
for i=0,6
  read jour(i)
next i
for i=1,12
  read mois(i)
next i
data Dimanche,Lundi,Mardi,Mercredi,Jeudi,Vendredi,Samed
 i
data Janvier,Février,Mars,Avril,Mai,Juin,Juillet
data Aout,Septembre,Octobre,Novembre,Décembre
print jour(time_d);time_n;" ";mois(time_mo);time_y
```

CLOCK [k]

NOCLOCK [k]

Variables d'état Visibilité de l'horloge

DATE [k]

NODATE [k]

Variables d'état Visibilité de la date

k

entier*32

Les deux variables d'état `clock` et `noclock` renvoient la même valeur entière x. Si $x = 0$ l'horloge est cachée, sinon l'horloge est affichée sur la case F40 du menu. Le nombre de bits non nuls de x règle le délai de réécriture de l'horloge. Par exemple $x =$ $FFFF correspond à une réécriture toutes les secondes, $x =$ $5555, à une toutes les deux secondes et $x = 1$ à une toutes les 16 s.

Pour faire apparaître (resp disparaître) l'horloge on peut utiliser la commande `clock` (resp `noclock`) sans préciser k. On peut aussi utiliser indifféremment une quelconque des deux commandes avec la valeur de k correspondant à l'état souhaité. Seule est prise en compte la valeur de $x = k$ `and` $FFFF.

Les deux variables d'état `date` et `nodate` renvoient la même valeur entière y qui a une signification analogue à x mais pour la date. Ainsi si $y = 0$ la date est cachée. Sinon la date est affichée devant l'heure dans le menu.

Pour pouvoir afficher l'horloge ou la date pendant l'exécution du programme, il est nécessaire que `cursh` soit plus grand ou égal à 4.

Exemple

Examiner, en mode direct, l'effet des commandes :

```
clock $5555
clock $1111
clock $0101
clock 1
clock 0
clock -1
```

Contrôle de l'exécution

NEW

Commande Efface la source

RUN fchaîne

RUN [ligne]

DEBUG [ligne]

Commandes Exécution

ligne

ligne de départ avec la même syntaxe que dans la commande `#ligne` de la fenêtre Basic.

La commande `new` efface la source et retourne à l'éditeur. Les commandes `run` et `debug` lancent le programme, comme les commandes du menu RUN et DEBUG ou RUN... et DEBUG... si la ligne de départ est précisée. Ces commandes du Basic sont les seules qui peuvent utiliser un numéro de ligne. La commande :

```
run "test"
```

vide la source, charge le fichier source "test.z" (l'extension par défaut est Z), et lance le programme. Avant cela, elle demande confirmation.

Exemple

Les commandes `run` et `debug` sont surtout utiles en mode direct. Dans la source, `run` *label* se comporte comme `goto` *label*. Noter que `goto r-3` est illégal alors que `run r-3` est admis. Plutôt que d'utiliser des instructions comme `run r-3` ou `run` 27 dans vos programmes, il vaut mieux placer des labels sur les lignes 27 et `r` − 3 et effectuer des branchements vers ces labels. La mise au point de vos programmes sera plus facile (imaginer que vous insériez une ligne devant la ligne `r`).

```
  x=1
  run r-3
  ------
  print x
  debug r
  ------
r:print x
```

Sortie (270 ms)

```
RUN
RUN
  1
  1
```

QUIT

Commande Sortie du Basic

La commande `quit` équivaut à la commande homonyme du menu TOOLS.

CLEAR

Commande Initialisation du programme.

La commande `clear` ne modifie pas la source. Elle efface tous les anciens noms et données en mémoire, et réinitialise les variables d'état graphiques (exceptée la variable `t_height`). Elle met de plus :

```
base ten
develop
```

```
precision 10
formatx 0
format -11
formatl 0
formatm 1
tilde
clear timer
clear cond
distingo
on error stop
on break stop
close
randomize 0
origin 0,0
```

Les seuls types implicites sont :

```
implicit char $
implicit index %,!
```

La commande `clear` n'est pas utilisable au milieu d'un programme en général. Comme chaque RUN (F8) ou DEBUG (F9) commence implicitement par `clear`, la commande est essentiellement utilisée en mode direct.

Exemple

Si `x` a été utilisé comme variable :

```
x=exp(-1/3)
```

avant de pouvoir utiliser `x` comme symbole, `clear` est nécessaire :

```
clear
print (x-3)^3
```

STOP

Commande Arrêt du programme

END

Commande Marque de fin du programme

La commande `stop` provoque le retour à la fenêtre Basic, mais les piles des procédures, fonctions, boucles restent en l'état. L'écran avant la commande `stop` est sauvegardé. Il sera rétabli par la commande éditeur Debug+.

Toutes les lignes après la commande `end` sont ignorées. L'exécution de la commande `end` est équivalente à `stop`. La commande `end` est active séparément dans la source et la bibliothèque.

Exemple

Le programme est correct. Il n'y a pas de label répété, par suite de la commande `end`.

```
  goto L
  print 1
L:print 2
```

```
    END
  L:etc.
```

Par contre :

```
    goto L
    etc.
    END
  L:etc.
```

est incorrect car il manque le label L.

BREAKPOINT

Commande Renvoie dans le débogueur

Si la commande `breakpoint` se trouve dans B_INIT ou B_TRACE, la commande n'est effective qu'après le retour du sous-programme.

Exemple

Le débogage commence ligne 2 après RUN.

```
breakpoint
etc.
```

REM ...

Commande Commentaire

On peut utiliser le signe ' (de code ASCII $27) au lieu de `rem`.

VERSION

Variable d'état Numéro de version

En tant que fonction, `version` renvoie un entier positif qui croîtra avec les mises à jour successives du Basic 1000d. En tant que commande, `version` permettra l'émulation des versions précédentes du Basic 1000d.

Exemple

La sortie indique que le Basic 1000d en est à sa première version.

```
print "Version=";version
```

Sortie (20 ms)

```
Version=  1
```

Traitement des erreurs et du Break

ON ERROR [GOTO] label

ON ERROR [STOP]

Commandes Traitement des erreurs

ERR

V_fonction Numéro d'erreur

ERL

V_fonction Ligne de l'erreur

ERA

V_fonction Adresse de la ligne de l'erreur

ERR$(k)

C_fonction Message d'erreur numéro *k*

ERROR k

Commande Simule l'erreur *k*

k

entier*16

Les mots clefs `goto` et `stop` dans les commandes `on error` sont facultatifs, et n'en modifie pas le sens. Après `on error` *label* une erreur provoque un `goto` *label* en modes Run ou Debug, sans sortie du message d'erreur. Si une nouvelle erreur se produit, elle provoque un arrêt à moins qu'une autre commande `on error` *label* n'ait été rencontrée.

Après `on error`, sans label, l'effet de la commande `on error` *label* est annulé. Une erreur provoque alors un arrêt du programme.

Les fonctions `err`, `erl` et `era` caractérisent la dernière erreur rencontrée. La commande `error` *k* simule l'erreur *k*.

La fonction `err$` renvoie le message d'erreur numéro *k*. Si *k* n'est pas un numéro d'erreur, elle renvoie `str$`(*k*).

Exemple

La deuxième ligne en erreur provoque un branchement vers la ligne `ex_err`.

```
       on error goto ex_err
       illegal
ex_err:print "Traitement d'erreur"
       print "Erreur numéro";err;"  (";err$(err);")"
       print "ligne";erl;"  (";peekz$(era);")"
       stop
```

Sortie (115 ms)

```
Traitement d'erreur
Erreur numéro  34  (INSTRUCTION ILLEGALE)
ligne  2  (illegal)
```

Exemple

Le programme suivant écrit la liste des messages d'erreur triée dans l'ordre alphabétique par `sort`.

```
nodistingo
char erm(150)
index t(150)
do
  i=i+1
  ift str$(i)=err$(i) exit
  erm(i)=err$(i)
loop
i=i-1
sort erm(1),i,1,t(1)
for j=1,i
  print justr$(t(j),3)f;erm(t(j))
next j
```

ON BREAK [GOTO] label

ON BREAK [STOP]

ON BREAK NEXT

Commandes Traitement du Break (Control, Shift et Alternate)
Les mots clefs `goto` et `stop` dans les commandes `on break` sont facultatifs, et n'en modifie pas le sens. Après `on break` *label* le Break provoque un `goto` *label* en modes Run ou Debug, au lieu d'interrompre le programme. Après `on break`, sans label, le Break provoque le retour à la fenêtre Basic. Après `on break next`, l'appui sur Control, Shift et Alternate est sans effet.

Exemple

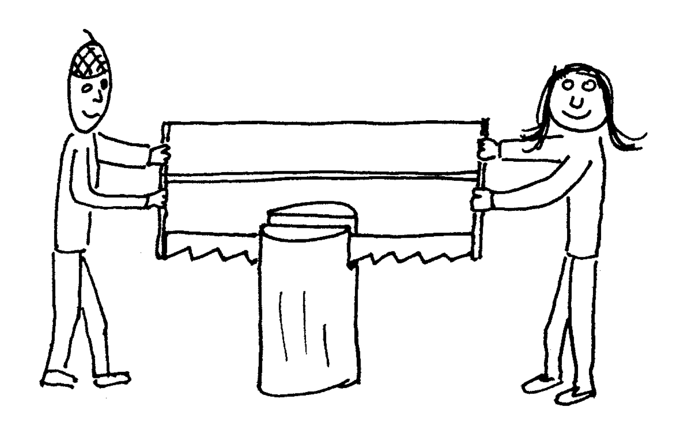
Pendant les 5 premières secondes, le Break est sans effet. Pendant les 5 secondes suivantes, il provoque un branchement en `b`. Sinon, pendant les 5 dernières secondes, il provoque le retour à la fenêtre Basic.

```
  on break next
  pause 5000
  on break goto b
  pause 5000
  on break stop
  pause 5000
  stop
b:print "Traitement du Break"
```

12

Labels

Sous-programmes



Labels et Branchements

GOTO label

Commande Branchement

Le branchement `goto` doit obligatoirement se faire par un label.

label (type de nom)

Pour le programme Basic, comme pour l'éditeur, les labels sont des noms non indicés, placés en tête de ligne et suivis de ":". Il y a cependant des différences entre le traitement des labels par l'éditeur et le Basic. Les labels de la bibliothèque sont inconnus de l'éditeur mais connus du Basic. Les labels de la source après `end` sont connus de l'éditeur, inconnus du Basic. Pour l'éditeur, il n'y a jamais de différences entre majuscules et minuscules, pour le Basic tout dépend la variable d'état `distingo` (ou `nodistingo`).

Exemple

Les majuscules sont identiques aux minuscules pour les labels `Lab2` et `Lab3`. Les majuscules sont différentes des minuscules pour les labels `Lab`, `LAB` et `LaB`. Tout dépend de la position du label par rapport aux instructions `distingo` et `nodistingo`, peu importe l'ordre d'exécution.

```
 Lab:print 1;
     goto LAB2
 LAB:print 2;
     goto Lab3
     nodistingo
Lab2:print 3;
     goto LAB
Lab3:print 4;
     distingo
 LaB:print 5
```

Sortie (55 ms)

```
 1  3  2  4  5
```

Fonctions et Procédures

Ce chapitre est assez difficile, par suite des nombreuses possibilités offertes par le Basic 1000d. Aussi, avant de nous lancer dans la description rigoureuse de la syntaxe, nous allons d'abord expliquer quelques exemples.

Récursion (Factorielle)

La fonction F(n) calcule $n!$ par récursion en s'appelant elle-même. Dans le programme de la fonction, le mot clef `value`, qui s'utilise comme une variable, permet de donner sa valeur à la fonction. Dans la fonction, `n` est une variable locale, qui n'influe pas sur les `n` des niveaux précédents (ni en particulier sur l'index `n` du programme principal). Il y a passage par valeur de l'argument.

```
    for n=1,40
      print n;F(n)
    next n
    stop
  F:function(n)
    value=1
    ift n=0 return
    value=n*F(n-1)
    return
```

Sortie (11410 ms)

```
 1  1
 2  2
 3  6
...
 40  815915283247897734345611269596115894272000000000
```

Passages d'arguments par valeur et adresse

L'appel de la procédure Z dans l'exemple suivant initialise 100 éléments d'un tableau avec les valeurs $U = 1200$, $U - 1 = 1199$, ..., $U - 99 = 1101$. Dans la procédure Z, les noms `i` et `T` sont des index locaux (`T` est initialisé avec la valeur 1200). Dans Z, U n'est pas une variable locale mais un nom local créé par `access` qui se comporte comme une variable indicée et permet d'accéder au tableau `T` du programme appelant. On a les correspondances :

appel	procédure
T(1)	U(0)
T(2)	U(1)
...	...
T(100)	U(99)

Après le retour de la procédure les noms `T` et `U` reprennent leurs anciens rôles. Le tableau `T`, passé par adresse est modifié, mais `U` qui a été passé par valeur reste inchangé.

```
    var T(100)
    U=1200
    Z T(1),U
    print U;conc$(i=1,100 of T(i))
```

```
        stop
      Z:procedure(access U(99),index T)
        local index i
        for i=0,99
          U(i)=T
          T=T-1
        next i
        return
```

Sortie (1355 ms)

```
 1200  1200  1199  1198  1197  1196  ...
```

Passage d'arguments par nom

Voici maintenant un exemple, où l'argument n'a pas une valeur, mais est un simple symbole. Dans la procédure, les `@1` sont remplacés par le symbole. C'est le passage d'arguments par nom.

```
        Sy sin
        Sy cos
        Sy exp
        stop
     Sy:print "@1(1)=";@1(1)
        return
```

Sortie (200 ms)

```
sin(1)=  0.8414709848~
cos(1)=  0.5403023059~
exp(1)=  0.2718281828~ E+1
```

Le passage d'arguments est en réalité toujours par nom, ce sont les instructions données dans le sous-programme qui permettent l'utilisation du passage d'arguments par nom comme un passage par valeur ou par adresse.

Structure des sous-programmes

Points d'entrée des sous-programmes

Les points d'entrée des fonctions et procédures sont les lignes comportant un label. Il y a trois types de labels : label de V_fonction, label de C_fonction et label/proc. Tous les labels peuvent être utilisés dans les branchements (`goto` label).

Exemple

Le programme affiche seulement le type des labels.

```
        type factori,chainep,procedu
```

```
        stop
  factori:function(n)
  chainep:function$(c(4))
  procedu:local index i
```

Sortie (160 ms)

```
 factori est de type label de V_fonction
 chainep est de type label de C_fonction
 procedu est de type label/proc
```

Selon l'instruction suivant le ":" après le label, tout label est de l'un des types ci-dessus. Si l'instruction est `function`, comme après `factori` dans l'exemple, (resp `function$`, comme après `chainep`) on a un label de V_ (resp C_) fonction. Si l'instruction suivant ":" est autre, comme après `procedu` on a un label/proc.

Les labels de fonctions, et seulement ceux-là, peuvent être utilisés comme FONCTIONS (avec ou sans arguments) dans les expressions. La valeur de la fonction est calculée, par un programme en Basic commençant par ce label, et se terminant par `return`.

Le programme de la fonction peut utiliser tout le Basic, y compris des appels de fonction à elle-même (récursivité). Nous étudierons dans les sections suivantes, comment on utilise les arguments, comment on assigne la valeur, les variables locales et autres possibilités.

Un label/proc peut être utilisé comme PROCEDURE. L'appel de la procédure, qui se fait par la commande [`gosub`] *label*, provoque alors l'exécution du programme commençant par le label. Après l'instruction `return` l'exécution reprend sur l'instruction suivant la commande d'appel (mais on dispose aussi de `return` *label* pour revenir ailleurs).

Nous étudierons les deux sortes de SOUS-PROGRAMMES (procédures et fonctions) en même temps par suite de leurs nombreuses ressemblances.

Appel des sous-programmes

Voici maintenant la syntaxe des appels de fonctions et procédures avec ou sans arguments. L'appel des fonctions est analogue à l'appel des fonctions internes du Basic.

nomfonc

nomfonc(listearg)

Appel des fonctions

nomfonc

label de C_ ou V_fonction

listearg

désigne la liste des arguments

Le label `nomfonc` peut être utilisé dans les expressions. Il provoque l'exécution du code Basic partant de la ligne du label. Nous verrons comment écrire ce code de

sorte qu'il calcule une valeur qui est utilisée dans l'expression à la place du nom `nomfonc`. L'appel peut transmettre *listearg*, qui est un texte quelconque tel que les parenthèses ouvertes "(" soient toutes refermées par ")", sans tenir compte des parenthèses entre guillemets (comme "((("). La séparation en arguments de listearg est étudiée plus bas.

Exemple

Le nom `ffact` dans la première instruction appelle la fonction `ffact`, qui calcule une valeur dépendant de l'argument `i`.

```
        print sum(i=0,12 of 1/ffact(i))
        stop
  ffact:function(n)
        value=float(ppwr(n))
        return
```

Sortie (315 ms)

```
 0.2718281828~ E+1
```

[GOSUB] label

[GOSUB] label(listearg)

[GOSUB] label listearg

Commande Appel des procédures

La liste des arguments *listearg*, même vide, peut être mise entre parenthèses, la parenthèse gauche "(" devant alors suivre *label* sans espace. Ces parenthèses sont facultatives. Le mot clef `gosub` peut être omis si *label* est du type label/proc.

Δ Vous pouvez déduire de cette règle qu'il est possible d'appeler par `gosub` (non omis) des labels de fonction. Dans ce cas il n'y a pas création de la variable locale `value`, qui reste donc définie à son niveau lors de l'appel.

La forme sans `gosub` et sans parenthèses autour de *listearg* est semblable à l'appel des commandes du Basic. L'appel doit obligatoirement se faire par un label et est impossible par un numéro de ligne. Le nombre de niveaux de sous-programmes est limité par la variable d'état `s_pro` (qui peut être modifiée si nécessaire). La longueur des listes d'arguments des divers niveaux d'appel est limitée par la variable d'état `s_xqt` (également modifiable).

RETURN [label]

Commande Retour de sous-programme

Pour les procédures, le retour se fait sur la ligne après l'appel, ou sur la ligne *label* si celui-ci est donné. Le retour de fonction se fait dans l'expression appelante, pour poursuivre le calcul de l'expression. La forme `return` *label* est interdite pour les retours de fonctions.

Exemple

P et Q sont des label/proc. Les instructions `P` $x + 10$ et `Q` sont des appels des procédures définies par ces labels. La procédure P revient sur la ligne

après son appel, Q revient sur la ligne R. L'argument $x + 10$ de l'appel de P est transmis à une variable locale `x` de P. L'exemple semble assez simple, cependant l'explication de la commande `procedure` ne pourra être donnée qu'en fin de ce chapitre.

```
    x=100
    P x+10
    print "Retour de P  x=";x
    Q
    ne retourne pas ici
  R:print "Retour de Q"
    stop
  P:procedure(x)
    print "Dans proc P  x=";x
    return
  Q:print "Proc Q"
    return R
```

Sortie (95 ms)

```
Dans proc P  x=  110
Retour de P  x=  100
Proc Q
Retour de Q
```

La commande `return` *label* est différente de `goto` *label*, en effet après `return` l'adresse de retour du sous-programme est dépilée.

Exemple

Comparez le programme (bien écrit) :

```
    for i=0,1000
      SP1
      ne passe pas ici
 R1:next i
    print "fin"
    stop
SP1:return R1
```

Sortie (1455 ms)

```
fin
```

et le programme suivant, incorrect, qui ne dépile pas les appels.

```
    do
      SP2
      ne passe pas ici
 R2:loop
    stop
SP2:goto R2
```

Sortie (215 ms)

```
*ERREUR* S_PRO TROP PETIT
```

```
?
2.SP2
```

Boucles et Sous-programmes

Sur l'instruction `return`, on termine les boucles (`for`, `forv`, `forc`, `do`, `repeat`, `while`) commencées dans la procédure ou fonction.

Exemple

```
          do
            P
          loop
        P:do
            return
          loop
```

s'exécute correctement.

Par contre, Basic 1000d ne diagnostique pas qu'un sous-programme termine des boucles commencées avant l'appel. Il continue la boucle dans le programme appelant, sans dépiler l'appel précédent.

Exemple

Utiliser le débogueur pour voir que le programme appelle continuellement Q sans dépiler les appels. Pour cela il faut revenir de temps en temps à l'éditeur et examiner la taille de la pile Proc par la commande =. Quand la pile est pleine il se produit la sortie erreur.

```
          do
            Q
          loop
        Q:loop
```

Sortie (120 ms)

```
*ERREUR* S_PRO TROP PETIT
?
2.Q
```

VALUE

Variable locale interne Valeur de la fonction

La variable `value` permet d'assigner sa valeur à une fonction. Un label de V_fonction peut prendre des valeurs exactes ou flottantes, suivant l'assignation de `value`. Un label de C_fonction prend pour valeur une chaîne de caractères.

Exemple

Ceci définit la fonction `rndf`, équivalente à la fonction interne `rnd`, qui renvoie une valeur aléatoire flottante entre 0 et 1. La fonction n'utilise pas d'arguments mais on peut l'appeler avec des arguments, comme dans le troisième appel. Ces arguments sont ignorés.

```
print rndf;100*(rndf-1/2);rndf(non pris en compte)
stop
```

```
     rndf:function
          value=random(2^precision2)*2~^-precision2
          return
```

Sortie (220 ms)

```
  0.3026658925~ E-1  0.1880871835~ E+2  0.7427453068~
```

Exemple

La fonction `spc3` renvoie la chaîne formée de 3 espaces. Si le nombre d'arguments, donné par `@0`, n'est pas nul, la fonction stoppe le programme.

```
          print "abc";spc3;"def"
          print spc3(1)
          stop
     spc3:function$
          if @0
            print "Arguments interdits"
            stop
          endif
          value="   "
          return
```

Sortie (20 ms)

```
abc   def
Arguments interdits
STOP
```

A l'entrée de la fonction, une variable locale de nom `value` est créée et initialisée à 0 (exact) pour les V_fonctions et à la chaîne vide pour les C_fonctions. Dans le corps de la fonction, `value` se comporte comme une variable locale, que l'on peut lire et modifier. Sa valeur est transmise à l'expression appelante après l'exécution de `return` et le programme continue ensuite le calcul de l'expression qui contient la fonction.

Si la fonction F appelle une procédure, la procédure a aussi accès à cette même variable `value`. Par contre, si la fonction F appelle la fonction G, la variable `value` de F n'est plus accessible dans G. On peut donc écrire, dans F :

```
          PPP value
```

qui transmet `value` à la procédure PPP, mais il est incorrect d'écrire :

```
          w=G(value+5)
```

car `value` dans l'argument sera dans G la variable `value` de la fonction G, pas celle de F. Ce comportement correspond au fait que les arguments ne sont pas calculés au moment de l'appel, mais seulement sur demande dans le sous-programme. Il faudra donc utiliser une autre variable :

```
          value_f=value
          w=G(value_f+5)
```

Ces explications sont résumées par la règle suivante :

Règle

NE PAS UTILISER `value` DANS LES APPELS DE FONCTIONS EXTERNES.

Par contre l'emploi de `value` dans les arguments de toutes les fonctions internes du Basic, par exemple `mod(value*x,x^2-2)` est correct.

L'emploi de `value` n'est pas autorisé si une fonction n'est pas en cours de calcul.

Les arguments

Envoi des arguments

Fonctions

Par exemple, les arguments A1, A2, ... sont transmis à la fonction XYZ par la notation XYZ(A1, A2, ...) où la liste des arguments séparés par des virgules est placée entre parenthèses derrière le nom du label de fonction XYZ. Dans l'appel :

```
print XYZ(a-b,print,+,)
```

il y a 4 arguments. En Basic 1000d, les arguments peuvent être presque n'importe quoi. Ici le premier argument ressemble à une expression, le deuxième est le mot clef `print`, le troisième le caractère + et le quatrième est vide.

Au moment de l'appel de la fonction, Basic 1000d ne cherche pas à calculer la valeur des arguments. Il détermine seulement le nombre et le texte des arguments, et ces données sont empilées pour permettre des appels récursifs. C'est au programme de la fonction de donner leur rôle aux arguments.

Procédures

Par exemple, les arguments A1, A2, ... sont transmis à la procédure PPP par la notation PPP A1, A2, ... où la liste des arguments séparés par des virgules est placée derrière le nom du label/proc PPP. La notation PPP(A1, A2, ...) est également admise.

Règles pour la séparation des arguments

Le contenu des zones entre guillemets est ignoré pour déterminer les arguments. Chaque argument contient autant de parenthèses gauches que droites. Aucun argument ne contient de "," non parenthésée. Les espaces en début et fin d'argument font partie de l'argument. Toutefois, dans le cas des procédures, les espaces devant le premier argument ne font pas partie de cet argument, et le dernier argument s'arrête éventuellement sur le caractère avant l'apostrophe indiquant un commentaire (cas de la liste des arguments non parenthésée).

Exemple

Les divers arguments sont indiqués sauf pour le dernier appel qui est incorrect parce que les parenthèses ne sont pas refermées.

```
print psi( A(4,8,9*C(4,8)) , "(((" & , ",")
         1111111111111111 222222222 3333
OVR   "I" ,#35 ,STEP.DAT'commentaire
      1111 2222 33333333
PPP  B(4,5 , C(3, 18  )
```

Utilisation des arguments

Nous avons vu qu'à l'appel du sous programme les arguments ne sont pas calculés. C'est le programme qui doit préciser l'utilisation des arguments. Les arguments peuvent être utilisés à la manière des arguments de macros dans les macroassembleurs. Dans le texte des instructions du sous-programme appelé, on peut introduire les arguments de l'appel. C'est l'instruction ainsi modifiée qui est exécutée.

En fait, toute instruction de la source comportant des caractères `@` est modifiée avant exécution, suivant les règles suivantes.

@ k [*f*]

Remplacement par l'argument *k*

k

primaire

k doit avoir une valeur entière ou nulle avant l'exécution de l'instruction.

Δ Le calcul de *k* ne doit pas appeler de fonctions écrites en Basic (mais il n'y a pas de vérification). En général ses valeurs sont purement numériques ou contiennent seulement un index, cette exigence n'est donc pas contraignante.

Si $k = 0$, `@`k est remplacé par le nombre d'arguments d'appel n. Par exemple si $n = 250$, `@0` est remplacé par §`250`. Si $k \in [1, n]$, `@`k est remplacé par le texte de l'argument k. Si k a une autre valeur, ou est absent, il y a erreur Valeur Après @. Si `@`k est dans une instruction du programme principal, il y a erreur @ Dans Main.

La partie de l'instruction qui est remplacée commence à `@` et se termine soit sur le caractère f ([a] S) si on a fait suivre k de ce caractère, soit sur le dernier espace après k.

Pour calculer la somme de tous les arguments, une instruction comme

```
w=sum(i=1,@0 of @i)
```

n'est pas correcte. En effet c'est avant l'exécution que l'on effectue le remplacement de `@i` par un argument (ce remplacement est effectué une seule fois). Si `i` a une valeur acceptable comme numéro d'argument, l'instruction calcule `@0` fois la valeur de cet argument, sinon erreur. On corrigera en :

```
w=0
for i=1,@0
  w=w+@i
next i
```

ou, en utilisant `arg$`, en :

```
w=sum(i=1,@0 of val(arg$(i)))
```

@@

Remplacement par `@`
Ce remplacement, effectué avant l'exécution de l'instruction, permet la présence du caractère `@`.

Exemple

Le premier appel de phi calcule sin 0.5 et le deuxième appel 4 cos 0.3. La fonction écrit également sur l'écran.

```
     print phi(sin,.5)
     print phi(4*cos,.3)
     stop
 phi:function
     print "(@@1 est remplacé par @1)";
     value=@1(@2)
     return
```

Sortie (245 ms)

```
(@1 est remplacé par sin)  0.4794255386~
(@1 est remplacé par 4*cos)  0.3821345957~ E+1
```

Exemple

On examinera le rôle de *f* ([a] S) dans cet exemple (utiliser le débogueur pour voir le texte modifié de l'instruction).

```
     print m10(-----<>----,%%%%^^^^^)
     stop
 m10:function$
     for i=1,@0
       value=value&"  @i ?" &  justl$(i) &"  @if ?"æ
     next i
     return
```

Sortie (75 ms)

```
-----<>----?1  -----<>---- ?
%%%%^^^^^?2  %%%%^^^^^ ?
```

Exemple

Le programme affiche la table ASCII comme la commande éditeur du menu HELPS. La fonction `F_1L` renvoie une chaîne de longueur 2 contenant l'écriture de l'argument. Comme la fonction est appelée en base 16, cette écriture est ici hexadécimale.

```
print /C/justc$("TABLE DES CODES ATARI",39)
print/H/"   ";conc$(I=0,$F OF F_1L(I))
FOR J=0,15
  print/H/F_1L(J);" ";
  videoinverse
  print/H/conc$(I=0,$F OF " "&chrp$(I*10+J))
  videonormal
```

```
        next J
        stop
    F_1L:function$
        VALUE=justr$(right$(@1,1),2)
        return
```

ARG$(k)

C_fonction Texte de l'argument *k*

La fonction `arg$` renvoie une chaîne de caractères contenant le texte de l'argument *k*. Si l'entier *k* ne correspond pas à un argument, elle renvoie la chaîne nulle.

Exemple

Ceci montre la différence entre le calcul de la chaîne `@1` et la chaîne `arg$`(1).

```
        px "AB" & "CD"
        stop
     px:print "arg$(1)=";arg$(1)
        print "@@1=";@1
        return
```

Sortie (40 ms)

```
arg$(1)="AB" & "CD"
@1=ABCD
```

Exemple

La fonction `arg$`(*k*) est utilisée pour détecter les arguments absents. Remarquer qu'il est nécessaire d'enlever les espaces aux extrémités de l'argument à l'aide de `justl$` avant de pouvoir affirmer son absence. Dans l'exemple `arg$`(1) est vide mais `arg$`(3) est une chaîne de deux espaces.

```
        py ,h ,  ,po
        stop
     py:for i=1,@0
          ift justl$(arg$(i))="" print "argument";i;"  absent"
        next i
        return
```

Sortie (70 ms)

```
argument  1  absent
argument  3  absent
```

Récursivité

Voici une fonction $F(N)$ qui calcule factorielle N par $F(N) = N\,F(N-1)$ si $N \neq 1$, et $F(1) = 1$:

```
    print F(5)
    stop
  F:function
    if @1=1
      value=1
    else
    R:value=(@1)*F(@1-1)
    endif
    return
```

Sortie (70 ms)

```
120
```

Quelles sont à votre avis, les instructions `R` exécutées (donc après substitution de `@1`) lors du calcul de $F(5)$? Réfléchissez bien, ce ne sont pas :

```
  R:value=(5)*F(4)
  R:value=(4)*F(3)
    ...
  R:value=(2)*F(1)
```

Vérifiez votre réponse en exécutant le programme à l'aide du débogueur (BREAK en R, puis plusieurs CYCL), ou en regardant la réponse un peu plus bas. Si votre réponse était bonne félicitations, vous avez tout compris. Si votre réponse était fausse, expliquer pourquoi on n'a pas écrit pour R :

```
    value=@1*F(@1-1)
```

et reprenez la question sur les instructions R, si on change R en :

```
  R:value=@1*F((@1-1))
```

La fonction $F(N)$ ne permet pas de calculer de très grandes factorielles, en effet les arguments `@1` occupent une très grande place et on obtient rapidement l'erreur S_xqt Trop Petit (i.e. la pile des arguments est trop petite). Voici une autre fonction $G(N)$, effectuant le même calcul, qui ne présente pas ce défaut, mais dont le fonctionnement est plus délicat à comprendre.

```
    print G(5)
    stop
  G:function
    i=@1
    if i=1
      value=1
    else
      value=i*G(i-1)
    endif
    return
```

Sortie (55 ms)

```
120
```

Examinez jusqu'à quelle valeur de N on peut appeler $G(N)$. La limite dépend de la dimension de la pile des appels, `s_pro`. La variable `i` de la fonction G n'est pas une variable locale, cependant le fonctionnement de G est correct, mais cela dépend de l'ordre de calcul des facteurs du produit `i*G(i-1)`. Cet ordre est de gauche à droite, comme dans toute opération du Basic 1000d.

Voici la réponse à la question sur les lignes R exécutées par le programme lors du calcul de F(5) dans l'exemple du début de cette section.

```
(XQT)R:value=(5)*F(5-1)
(XQT)R:value=(5-1)*F(5-1-1)
(XQT)R:value=(5-1-1)*F(5-1-1-1)
(XQT)R:value=(5-1-1-1)*F(5-1-1-1-1)
```

REMEMBER n

Commande Mise en mémoire de la valeur d'une fonction

n

entier $|n| < 2^{30}$

La commande `remember` utilise une table interne qui à `s_rem` entiers associe la valeur d'une V_ ou C_fonction. Le mot clef `s_rem` désigne une variable d'état qui permet de modifier la dimension de cette table. La commande `remember` permet d'accélérer la vitesse des fonctions, en évitant le recalcul des `s_rem` dernières valeurs. Un exemple spectaculaire, le calcul de la suite de Fibonacci, a été donné dans le chapitre de présentation.

Exemple

La fonction $F(N)$ suivante calcule factorielle N par récurrence. L'effet du programme est identique si on supprime la commande `remember`, mais le programme prend alors 230 s au lieu de 4 s. De plus il faut augmenter la pile des appels par :

```
s_pro 20000
```

L'accroissement de la vitesse est donc remarquable. On notera que ce calcul des factorielles de 1 à 200 est même plus rapide que par la fonction interne `ppwr`, qui s'effectue en 21 s.

```
  for I=1,200
    W=F(I)
  next I
  print F(200)
  stop
F:function(index I)
  remember I
  if I=1
    value=1
  else
    value=F(I-1)*I
```

```
        endif
        return
```

Sortie (4 s)

```
 788657867364790503552363213932185062295135977687173263294742533244359
 449963403342920304284011984623904177212138919638830257642790242637105
 061926624952829931113462857270763317237396988943922445621451664240254
 033291864131227428294853277524242407573903240321257405579568660226031
 904170324062351700858796178922222789623703897374720000000000000000000
 00000000000000000000000000000
```

Lorsque la commande :

```
        remember I
```

est exécutée, il y a deux possibilités. Tout d'abord si `I` n'est pas une entrée de la table du `remember`, la fonction est évaluée normalement, et sa valeur remplace dans cette table, la valeur la plus ancienne. Si par contre `I` est une entrée de la table du `remember`, la valeur de la fonction est simplement lue dans la table. La table du `remember` est vidée par `clear`.

Il est possible d'utiliser `remember` simultanément dans plusieurs fonctions. Il faut alors des entrées distinctes. Par exemple si les arguments de $F(n)$ et $G(n)$ sont limités à [0,100], on peut écrire :

```
      F:function(n)
        remember n
        ...
      G:function(n)
        remember 101+n
        ...
```

Exemple : Calcul du nombre de partitions

Ce problème de base de la théorie additive des nombres, nous fournit un exemple de `remember`. Par exemple il y a 5 partitions du nombre 4 :

$$4 = 3 + 1 = 2 + 2 = 2 + 1 + 1 = 1 + 1 + 1 + 1$$

La fonction $p(n)$ suivante calcule le nombre de partitions de n par la formule de récurrence d'Euler. La valeur exacte $p(n)$ est comparée à la valeur $R(n)$ d'une formule approchée de Rademacher (1937). Les deux premières instructions modifient la structure du Basic : `s_pro` permet une profondeur d'appel importante, et on donne à `s_rem` sa valeur optimum 201 (aucune valeur de $p(n)$ ne sera calculée deux fois).

```
        s_pro 20000
        s_rem 201
        format 1
        forv i in (4,[10,40,5],[50,200,50])
          print using "p(###)=",i;
          print using "# ### ### ### ###",p(i);
          print using "    R=# ### ### ### ###. ",exp(pi*sqr(2~
```

```
      *i/3))/4/i/sqr(3)
    nextv
    stop
  p:function(index n)
    remember n
    if n=0
      value=1
    else
      local index k
      do
        k=k+1
        ift n-(3*k^2-k)/2<0 exit
        vadd value,(-1)^(k+1)*p(n-(3*k^2-k)/2)
      loop
      k=0
      do
        k=k+1
        ift n-(3*k^2+k)/2<0 exit
        vadd value,(-1)^(k+1)*p(n-(3*k^2+k)/2)
      loop
    endif
    return
```

Sortie (97 s)

```
p(  4)=               5    R=                6.~
...
p(100)=     190 569 292    R=      199 280 893.~
p(150)=  40 853 235 313    R=   42 369 336 269.~
p(200)=3 972 999 029 388    R=4 100 251 432 187.~
```

Noms locaux

LOCAL {elocal}

Commande Déclaration et initialisation d'éléments locaux

elocal

```
  DATAV expr {,expr}
| DATAC exprchaîne {,exprchaîne }
| DATAI entier*32 {,entier*32 }
| DATAA nomi {,nomi}
```

```
  | VAR nomi {,nomi}
  | CHAR nomi {,nomi}
  | INDEX*size nomi {,nomi}
  | LIT nomi {,nomi}
  | ACCESS nomi {,nomi}
```

Chaque elocal commence par un mot clef qui indique son rôle. Les mots clefs déjà familiers `var`, `char`, `index` et `lit` correspondent simplement à la déclaration de variables, index et littéraux locaux. Les elocaux `datav`, `datac` et `datai` donnent des valeurs initiales. Les elocaux `dataa` et `access` permettent en quelque sorte de rebaptiser des variables ou index. Avant de poursuivre les explications, voici d'abord la commande inverse de `local`.

NOLOCAL

Commande Suppression des éléments locaux du niveau courant de sous-programme

Niveau de sous-programme

Dans le programme principal ou après `clear`, c'est 0. Après un appel de fonction ou de procédure le niveau augmente de 1, après `return`, il diminue de 1. L'erreur Return Sans Appel se produit si le niveau devient négatif. Noter qu'après retour à l'éditeur, par exemple pendant le débogage, le niveau n'est pas modifié (les retours de procédures sont possibles après reprise du programme).

Il est possible d'utiliser les commandes `local` et `nolocal` au niveau zéro de sous-programme.

Propriétés des éléments locaux

Le nombre de variables locales utilisables est limité par la variable d'état `s_var` (qui est modifiable). Le nombre d'index locaux est lui limité par la variable d'état `s_pro`. Les index locaux, comme les index globaux, occupent une place fixe en mémoire.

Exemple

Cet exemple montre ce qui se passe lorsqu'on crée un élément local ayant un nom déjà utilisé par le programme. L'ancien objet n'est plus accessible par le programme. On retrouve cependant l'ancien objet après les commandes `nolocal` ou `return`.

```
char W
W="abcd"
local index W
type W
print "W=";W
nolocal
type W
print "W=";W
```

Sortie (110 ms)

```
W est de type index
```

```
W=  0
W est de type char
W=abcd
```

Attention, il n'existe pas de labels locaux en Basic 1000d. Il n'est pas possible d'utiliser des noms de labels ou des mots clefs comme variables locales.

Exemple

La deuxième commande qui essaie de créer une variable locale ayant le nom d'un élément local du même niveau de sous-programme produit une erreur Local.

```
     local index F
     local char F
```

Sortie (70 ms)

```
*ERREUR* LOCAL
local char ?
2.local char F
```

Littéraux

Le cas des noms de littéraux est un peu délicat.

Exemple

Dans le programme principal `x` est un littéral qui rentre dans la variable `W`. Dans la procédure SP, `x` est utilisable sans problème comme variable, on lui donne la valeur de `W`. Cependant, à l'impression de `W`, on voit réapparaître le littéral `x`. Dans SP, toutes les utilisations de `x` concernent la variable locale `x`. Ainsi `print deg(W, x)` y produirait une erreur.

```
     W=(1+x)^2
     SP
     type x
     stop
  SP:local var x
     x=W
     type x
     print "x local=";x
     return
```

Sortie (140 ms)

```
x est de type var
x local=  x^2 +2*x +1
x est de type lit
```

L'utilisation de littéraux locaux, elle aussi, peut produire des phénomènes étonnants.

Exemple

Dans cet exemple on utilise un littéral local pour donner une valeur à la variable globale `W`. De retour dans le programme principal, on obtient l'erreur Local en voulant écrire `W`.

La différence avec l'exemple précédent provient du fait que les noms d'un niveau inférieur ne sont pas effacés (il faut pouvoir les retrouver), tandis qu'après `nolocal` ou `return` les noms locaux sont perdus. On a donc autorisé l'impression des littéraux cachés, avec leur nom caché, et interdit l'impression de littéraux locaux effacés.

```
      SP
      type W
      print "W=";W
   SP:local lit x
      W=(1-x)^2
      print "W=";W
      return
```

Sortie (135 ms)

```
W=  x^2 -2*x +1
W est de type var
*ERREUR* LOCAL
print "W=",W?
3.print "W=",W
```

Exemple

Le littéral local `y` de SP2 possède le même numéro (les littéraux locaux sont numérotés à partir de $7FFF en décroissant) que le littéral local `x` de SP1 utilisé pour créer `W`. La variable `W` s'écrit dans SP2 avec le nouveau nom `y`.

```
      SP1
      SP2
      stop
  SP1:local lit x
      W=1/(1+x)
      return
  SP2:local lit y
      print W
      return
```

Sortie (25 ms)

```
[y +1]^-1
```

Les valeurs initiales

Il est possible dans l'instruction `local` de donner des valeurs initiales, à l'aide des elocaux `datav`, `datac`, `datai` et `dataa`. Si on ne donne pas de valeurs initiales, les index et variables sont initialisés à 0 (exact) ou à la chaîne vide.

La commande `local` construit deux listes différentes à partir des données. Une liste d'entier*32 à partir des elocaux `datai` et `dataa`, et une liste de valeurs à partir de `datav` et `datac`. Les index déclarés dans les elocaux `index` sont initialisés, dans l'ordre, avec les entier*32 de la première liste, jusqu'à épuisement de la liste. Cette première liste sert également à la définition des

`access`, mais nous étudierons cela un peu plus tard. S'il reste des données dans la liste, l'erreur Trop De Données est générée.

Exemple

La commande `local` définit les index locaux `i` et `j(,)`, et les initialise avec les données, sauf `j`(1,1) qui est initialisé à 0. Noter que les trois elocaux ne sont pas séparés par des virgules.

```
local datai 1,2,3,4 index i index*4 j(1,1)
print "i=";i
print "j(,)=";j(0,0);j(1,0);j(0,1);j(1,1)
```

Sortie (70 ms)

```
i=  1
j(,)=  2  3  4  0
```

Les variables des types var et char sont traitées simultanément. Attention à l'ordre inverse des tableaux (voir `var`, `char`).

Exemple

L'ordre d'initialisation des tableaux de variables est l'opposé de celui des tableaux d'index.

```
local datav 1,2,3,4 var a,b(1,1)
print "a=";a
print "b(,)=";b(1,1);b(0,1);b(1,0);b(0,0)
```

Sortie (70 ms)

```
a=  1
b(,)=  2  3  4  0
```

On peut assigner à une variable de type char le contenu d'une expr, mais l'assignation d'une chaîne à une variable de type var n'est en général pas possible.

Exemple

L'exemple revient à initialiser `c` avec `mkx$`(1). La conversion de `c` en expr est ensuite réalisée avec `push$` et `pop`.

```
local datav 1 char c
push$ c
print pop
```

Sortie (15 ms)

```
 1
```

On peut mélanger données et déclarations.

Exemple

```
local datai 1,2 index i datai 3 index j,k
print i;j;k
```

Sortie (40 ms)

```
 1  2  3
```

On peut aussi mélanger `var`, `char` et `index`.

Exemple

```
        local datai 1,2,3 datav 4,5 datac "a" index i,j,k var p
         ,q char c
        print c;i;j;k;p;q
```

Sortie (50 ms)

```
 a  1  2  3  4  5
```

Global ou local?

Exemple très important

Avant de lire les explications, essayez de comprendre par vous-mêmes la sortie de ce programme, en l'exécutant par le débogueur.

```
        A=120
        SP A
        stop
     SP:local var A
        A=@1
        print A
        return
```

Sortie (15 ms)

```
   0
```

L'instruction `A=@1` est exécutée sous la forme `A=A`. Le nom `A` aussi bien à droite qu'à gauche de "=" est le `A` local à la procédure qui n'a plus rien à voir avec le `A` du programme principal. L'apparente complexité de l'instruction `local` permet de transmettre aux variables locales des valeurs calculées avec les variables du niveau précédent. Ainsi, dans :

```
        A=120
        SPM A
        stop
    SPM:local datav @1f var A
        print A
        return
```

Sortie (15 ms)

```
   120
```

la variable locale `A` est initialisée avec la valeur de la variable globale de `A`, parce qu'au moment du calcul de `@1`, la variable locale `A` n'a pas encore été déclarée. Remarquer qu'il faut écrire `@1`*f* ou `(@1)` pour éviter une erreur de syntaxe dans `local`. Autrement dit, la ligne SPM ci-dessus est équivalente à :

```
    SPM:push @1
        local var A
        A=pop
```

Exemple

Voici maintenant un exemple analogue pour la sortie.

```
        P A,45
```

```
    print A
    stop
  P:local datav @2f var A
    @1=A
    return
```

Sortie (15 ms)

```
A
```

L'assignation `@1=A` est exécutée sous la forme `A=A`, mais en terme de la variable locale `A`. Pour faire sortir la valeur, voici une méthode utilisant uniquement les notions exposées jusqu'ici.

```
    P A,45
    print A
    stop
  P:local datav @2f var A
    push A
    nolocal
    @1=pop
    return
```

Sortie (20 ms)

```
45
```

La valeur à renvoyer est calculée de façon locale, puis poussée dans la pile en fin de procédure. La commande `nolocal` est effectuée, ce qui permet de retrouver les variables globales. L'assignation `@1=pop` (en clair `A=pop`) a lieu avec la variable `A` du niveau précédent.

Noter que dans le cas présent, on pouvait utiliser pour P :

```
  P:@1=@2
    return
```

DATAA ... ACCESS ...

Nous examinons maintenant les elocaux `dataa` et `access`. Ils permettent de définir des noms locaux accédant à des variables et index des niveaux précédents.

Exemple

La commande `local` permet de définir un nom local `cl` qui sert de pseudonyme de la variable globale `c`. Nous dirons que `cl` est un accès de type char. Après l'assignation `cl="alpha"` la variable globale `c` contient “alpha”.

```
    char c
    local dataa c access cl
    cl="alpha"
    nolocal
    print c
```

Sortie (30 ms)

```
alpha
```

Les nomi des listes `dataa` et `access` se correspondent, et le nom local créé est du même type que le nom global auquel il accède. Dans l'exemple ci-dessus, il est nécessaire de spécifier le type de `c` avant l'appel de `local`, et `cl` est du type char comme `c`. Cependant, les noms peuvent être indicés de façons différentes.

Exemple

```
var v(100)
index j(2,5)
local dataa v,j(0,5),v(8) access vl(5,5),jl(2),vp
```

La variable locale `vl` possède 2 indices, et correspond aux variables globales suivant la correspondance :

```
v(0)  vl(0,0)
v(1)  vl(1,0)
...
v(6)  vl(0,1)
...
v(35) vl(5,5)
```

Remarquer que dans `dataa` on a écrit `v` sans indice mais on aurait pu écrire `v`(0) ou `v`(`min`) avec la même signification. L'index local à un indice `jl` accède à l'index global `j` comme suit :

```
j(0,5) jl(0)
j(1,5) jl(1)
j(2,5) jl(2)
```

La variable locale non indicée `vp` accède à la variable globale `v`(8).

L'utilisation d'`access` permet d'éviter les problèmes de conflits de noms en sortie comme celui de l'exemple que nous avons vu un peu plus haut. Voici comment on peut récrire cet exemple très simplement avec `access`. Le type de `A` doit être connu lors de l'exécution de `dataa` A. On a donc rajouté la première ligne.

```
  var A
  P A,45
  print A
  stop
P:local dataa @1f datav @2f access Y var A
  Y=A
  return
```

Sortie (15 ms)

```
45
```

Le Basic permet également des accès avec changement de type, et également des accès de type index en une adresse quelconque de la mémoire. Cela permet des effets intéressants, comme par exemple de créer un accès index*1, `bit`(31) sur un index*32, le tableau `bit`(31) permettant de manipuler séparément les bits de l'index.

Δ Ces accès spéciaux sont créés par `datai` au lieu de `dataa`. L'elocal `dataa nomi` est équivalent à l'elocal `datai $31303030,nb_max,a,t` qui met dans la liste des données entières du `local` quatre valeurs. La première valeur, $31303030 sert seulement de vérification, pour éviter un mauvais mélange des elocaux `index` et `access`. La deuxième valeur, nb_max, indique le nombre maximum d'éléments permis pour l'accès correspondant, s'il est indicé. En effet, l'accès local n'est pas autorisé à dépasser les bornes du tableau auquel il accède. La quatrième valeur, t, indique le type de l'accès :

t	type
$30020	index*32
$10020	index*16
$20	index*8
$4010020	index*4
$6020020	index*2
$7030020	index*1
$30	var
$40	char

Δ Lorsque le type est un index, la troisième valeur a est l'adresse de l'index (=`ptr`(`nomi`)). Lorsque le type est char ou var, a=`varnum`(`nomi`) est le numéro de la variable.

Exemple

Création de l'index*32 `c200hz` accédant à l'adresse $4BA
La lecture de `c200hz` est équivalente à `peekls`($4BA). On peut écrire :

```
c200hz=0
```

qui est équivalent à :

```
pokels $4BA,0
```

Nota : modifier le compteur en $4ba (ou `c200hz`) peut provoquer des ennuis disques, parce que ce compteur est utilisé par le système pour déterminer le changement de lecteur.

```
local datai $31303030,1,$4BA,$30020 access c200hz
t=c200hz
W=(1+x)^100
print c200hz-t
```

Sortie (2060 ms)

```
405
```

Exemple

Création d'un accès index*16 `word`(1) accédant aux mots haut et bas de l'index `i` et d'un accès `bit`(31) de type index*1 accédant aux 32 bits de `i`.
On écrit les mots haut et bas de `i` par `word`(0) et `word`(1), puis les bits correspondant au mot haut. Remarquer que c'est `-bit`(31-j) qui donne le bit_j du mot long `i`. Le mot haut est modifié par `word`(0)=1, puis par `bit`(15)=0.

```
index i
local datai $31303030,2,ptr(i),$10020 access word(1)
local datai $31303030,32,ptr(i),$7030020 access bit(31)
i=random(2^31)
print/h/ i;word(0);word(1)
print/B/word(0);conc$(j=0,$F of -bit(j))
word(0)=1
print /h/i
bit(15)=0
print /h/i
```

Sortie (220 ms)

```
2F21185C  2F21  185C
1011100100001  0  0  1  0  1  1  1  1  0  0  1  0  0  0  0  1
1185C
185C
```

Restriction dans le cas des index*1, *2 et *4.

Les accès doivent partir du bit_7 d'un octet, sinon l'erreur Non Entier est générée.

Exemple

Erreur, car `S`(1) commence sur le bit_3 d'un octet.

```
index*4 S(10)
local dataa S(1) access U
```

Sortie (90 ms)

```
*ERREUR* NON ENTIER
local dataa S(1?
2.local dataa S(1) access U
```

Déclaration simplifiée des éléments locaux

PROCEDURE(liste)

ARGUMENT(liste)

FUNCTION(liste)

FUNCTION$(liste)

Commandes Déclaration de variables locales et passage d'arguments

liste

```
[ vci ] nomi { [,] [ vci ] nomi}
```

vci

```
  var
| char
| index*size
| access
```

Dans *liste*, vci indique le type du nom qui suit. Si vci est omis, on prend le type du dernier vci donné, ou le type var si aucun vci n'a été donné. Les nomi sont définis par ces commandes comme étant des variables (de types var ou char), accès ou index**size* locaux (éventuellement indicés). Ces nomi sont initialisés avec les arguments `@1`, `@2`, ... pris dans l'ordre croissant.

En fait, ces commandes sont équivalentes à une commande `local`, et sont d'ailleurs vraiment effectuées par une commande `local`, comme on peut s'en rendre compte avec le débogueur.

Exemple

```
  T 1,2,3,4,5,6,7,8,9
  stop
T:procedure(A,B(2),char C(4))
  argument(index*8 D(5))
  function(index E,F(1+@1))
  print A;B(0);B(1);B(2);C(0);C(1);C(2);C(3);C(4)
  print conc$(i=0,5 of D(i))
  return
```

Sortie (140 ms)

```
1  4  3  2  9  8  7  6  5
1  2  3  4  5  6
```

La commande `procedure` équivaut à l'instruction :

```
local datav @1,@2,@3,@4f datac @5,@6,@7,@8,@9f
 var A,B(2) char C(4)
```

Elle définit la variable locale `A`, initialisée avec le premier argument, les tableaux de variables `B`(2) et C(4). Attention à l'ordre inverse d'initialisation. `B`(2) est initialisé avec l'argument 2, `B`(1) avec l'argument 3 et `B`(0) avec l'argument 4.

La commande `argument` définit un tableau d'index locaux `D`(5), initialisé dans l'ordre croissant (`D`(0) avec l'argument 1, ... `D`(5) avec l'argument 6). Il est possible, comme ici, de ne pas utiliser tous les arguments. Il est par contre impossible de donner plus de nomi que d'arguments.

Au lieu du mot clef `argument` on aurait pu utiliser les mots clefs `procedure`, `function` ou `function$` avec le même effet. Quatre mots clefs pour la même commande peut sembler superflu, mais cela permet d'améliorer la lisibilité

des programmes. En effet, on peut réserver `procedure` aux débuts de procédures (La commande `procedure` nue, inactive, sans liste ni (), est également acceptée). De même on utilisera `function` et `function$` seulement en début de V_ ou C_fonction, où une telle commande, éventuellement nue est de toutes façons obligatoire. `argument` sera utilisé dans le corps des procédures ou fonctions.

Considérons maintenant la commande `function`(*liste*) dans la procédure T de l'exemple ci-dessus. On remarque qu'il est possible d'utiliser des arguments (ici `@1`) dans cette commande. Cette commande prépare son `local` implicite en décodant d'abord, dans *liste*, les indices des tableaux, s'il y en a. Autrement dit, l'instruction compte d'abord le nombre d'arguments représentés par *liste*. Dans l'exemple l'indice `1+@1` va donc être calculé (=2). Par contre il aurait été incorrect d'écrire :

```
function(index E,F(1+E))
```

car au moment du calcul de 1+E, le programme ne connaît pas encore E.

Exemple

Reprenons l'exemple de la procédure P qui assigne le deuxième argument au premier. Il est maintenant on ne peut plus simple.

```
  var A
  P A,45
  print A
  stop
P:procedure(access Y,var A)
  Y=A
  return
```

Sortie (15 ms)

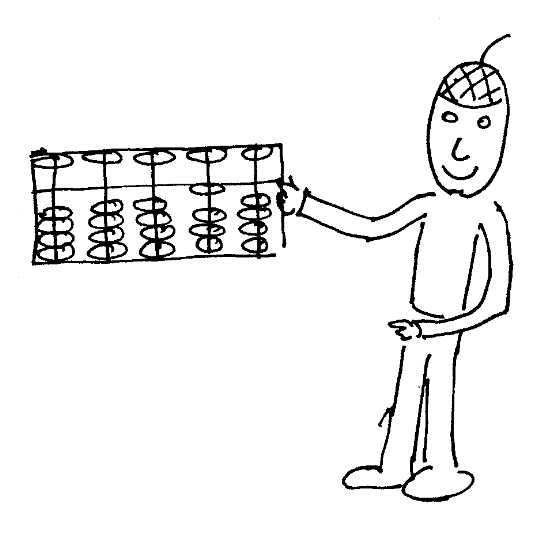
```
45
```

Exercice Déterminant

Nous avons déjà donné une fonction (dete) qui calcule un déterminant à partir de la liste de tous ses éléments. Ecrire une autre solution, en utilisant la commande `function` pour initialiser un tableau indicé local avec la liste d'appel.

13

Calculs exacts



Les fonctions étudiées dans ce chapitre effectuent des calculs exacts, sans approximations. Cependant, certaines de ces fonctions (comme `min` ou `abs`) renvoient un résultat approché en flottant si un des arguments est flottant. On remarquera que les fonctions implicites + − ∗ / `mod` et `div` dans les expressions se comportent de même, en effectuant des calculs exacts ou flottants suivant le type des arguments.

Fonctions numériques

Nous examinons ici principalement des V_fonctions qui renvoient un nombre exact.

EXACT(p)

V_fonction Conversion en forme exacte

p

expr

La fonction `exact` convertit un nombre flottant p en une forme exacte, sans perte de précision. La valeur obtenue pour p réel, $A2^{\alpha}$ avec A et α entiers, correspond au codage des flottants. La fonction accepte également une expression exacte quelconque comme argument p. Elle renvoie alors cet argument sans modification.

Exemple

Le programme suivant détermine l'erreur relative sur le calcul du nombre complexe e^{1+i} en précision 10. Pour cela, ce nombre est calculé en précision 20 et converti en exact. La valeur w ainsi obtenue est ensuite considérée comme étant la valeur exacte pour calculer l'erreur relative.

```
complex i
precision 20
w=exact(exp(1~+i))
precision 10
format -5
print cabs((w-exact(exp(1~+i)))/w)
```

Sortie (1535 ms)

```
0.9773~ E-14
```

APPR(p [, k])

V_fonction Meilleure fraction approchée

p

expr (exacte ou nombre flottant)

k

entier > 0 (par défaut $k =$ `precision2`)

Si p est réel, la fonction `appr` renvoie la fraction rationnelle f la plus simple possible qui soit une approximation à mieux que 2^{-k} de p ($|p - f| < 2^{-k}$). La méthode consiste à développer p en fraction continue jusqu'à la précision voulue.

Dans le cas où p est un polynôme développé, cette approximation est effectuée sur tous les coefficients du polynôme.

Dans le cas où p est une expression factorisée, cette approximation est effectuée sur le facteur numérique `factorp`(p, 1) (les autres nombres dans p sont entiers et ne changent pas). En général, si p est factorisé `appr`(p) et `appr`(`formd`(p)) diffèrent.

Exemple

Calcul des convergents de π. Cet exemple montre aussi la différence entre `exact` et `appr`.

```
precision 20
print "pi=";pi;"  est représenté par"a;exact(pi)
format -5
forv i in (1,3,10,14,22,31)
  w=appr(pi,i)
  print justl$("appr(pi,"&justl$(i,2)&")="&w,30);
   "erreur=";abs(w-pi)
nextv
```

Sortie (875 ms)

```
pi=  0.31415926535897932385~ E+1  est représenté par
   124451306656115542615260972311/39614081257132168796771975168
appr(pi,1 )=  3               erreur=  0.1416~
appr(pi,3 )=  22/7            erreur=  0.1264~ E-2
appr(pi,10)=  333/106         erreur=  0.8322~ E-4
appr(pi,14)=  355/113         erreur=  0.2668~ E-6
appr(pi,22)=  103993/33102    erreur=  0.5779~ E-9
appr(pi,31)=  104348/33215    erreur=  0.3316~ E-9
```

Exemple

Cet exemple illustre la différence d'action sur les expr développées et factorisées.

```
w=sum(i=1,5,4 of random(2^20)/random(2^20)*x^i)
print "w=";w
print "appr(w,15)=";appr(w,15)
factor
w=w
print "formf(w)=";w
print "appr(formf(w),15)=";appr(w,15)
```

Sortie (335 ms)

```
w=  274591/240732*x^5 +777446/135101*x
```

```
appr(w,15)=  73/64*x^5 +633/110*x
formf(w)=  1/32523133932* [x]* [37097518691*x^4 +187156130472]
appr(formf(w),15)=  0
```

Exemple

Pour l'approximation d'un nombre complexe flottant, il faut être en option `develop`. Le programme ci-dessous en option `factor` donnerait le résultat 0, pour une raison similaire au résultat 0 de l'exemple précédent.

```
complex i
w=pi+i/pi
print appr(w,30)
```

Sortie (250 ms)

```
33102/103993*i +103993/33102
```

SGN(p)

V_fonction Signe

p

réel

La fonction `sgn` renvoie la valeur exacte :

p	sgn(p)
$p < 0$	-1
$p = 0$	0
$p > 0$	1

Exemple

```
print sgn(-1/19);sgn(0);sgn(pi^2)
```

Sortie (40 ms)

```
-1  0  1
```

ABS(p)

V_fonction Valeur absolue

p

réel

Lorsque l'argument p est flottant, le résultat de la fonction `abs` est également en flottant. La valeur absolue en complexe est calculable par `cabs` ou `cxabs`, mais pas par la fonction `abs`.

Exemple

```
print abs(-7/3);abs(7~/3)
```

Sortie (45 ms)

```
7/3  0.2333333333~ E+1
```

MIN(p, q {, r})

MAX(p, q {, r})

V_fonctions Minimum et maximum

p, q, r

réel

Lorsqu'un des arguments est flottant, le résultat est également en flottant. Les mots clefs `min` et `max` sans arguments sont utilisés avec d'autres sens pour l'accès des tableaux, et (seulement `max`) dans les commandes `himem` et `limit`. Dans le cas des chaînes, les fonctions analogues ont des noms différents, `min$` et `max$`.

Exemple

Le programme écrit le nombre médian parmi trois.

```
a=-7
b=45
c=3
print a+b+c-min(a,b,c)-max(a,b,c)
```

Sortie (25 ms)

```
3
```

FIX(p)

V_fonction Partie entière de *p*

CINT(p)

V_fonction Plus proche entier de *p*

LINT(p)

INT(p)

V_fonction Plus grand entier inférieur ou égal à *p*

GINT(p)

V_fonction Plus petit entier plus grand ou égal à *p*

p

réel

Les fonctions `lint` et `int` sont synonymes. La table suivante compare les fonctions `fix`, `cint`, `int` et `gint`.

p	`fix`	`cint`	`int`	`gint`
2.6	2	3	2	3
2.4	2	2	2	3
2	2	2	2	2
−2	−2	−2	−2	−2

Le réel p est un entier si et seulement si une quelconque de ces fonctions renvoie la valeur p. La fonction `cxint` est l'analogue de la fonction `cint` pour les nombres complexes.

Exemple

Montre comment obtenir les nombres de la table ci-dessus.

```
p=-2.4
print fix(p);cint(p);int(p);gint(p)
```

Sortie (50 ms)

```
-2 -2 -3 -2
```

EVEN(x)

ODD(x)

V_fonctions Parité

x

entier

La fonction `even` (resp `odd`) renvoie -1 (`true`) si x est pair (resp impair) et 0 (`false`) sinon.

Exemple

```
x=2^50+random(2)
print odd(x);even(x)
```

Sortie (20 ms)

```
-1  0
```

NUMR(p)

DENR(p)

V_fonction Numérateur et dénominateur

p

réel exact

Si $p = a/b$ où a et b sont des entiers premiers entre eux ($b > 0$), les entiers a et b sont renvoyés par les fonctions `numr` et `denr` respectivement.

Exemple

```
print "123/456=";numr(123/456);"/";denr(123/456)
```

Sortie (45 ms)

```
123/456=  41/  152
```

INTSQR(p)

V_fonction Partie entière de la racine carrée

p

réel $p \geq 0$

La précision de la fonction `intsqr` est infinie (quel que soit `precision2`). L'entier $k =$ `intsqr`(p) vérifie donc toujours les inégalités :

$$k^2 \leq p < (k+1)^2$$

Exemple

Le calcul de `sqr`(w) dépend de la précision, à la différence de celui de `intsqr`(w).

```
w=(10^25)^2-10^-100
print intsqr(w);int(sqr(w))
```

Sortie (145 ms)

```
9999999999999999999999999  9999999999999998758486016
```

INTROOT(p, k)

V_fonction Partie entière de la racine k-ième

p

réel $p \geq 0$

k

entier $k \in [1, 2^{15}[$

La précision de la fonction `introot` est infinie (quel que soit `precision2`). L'entier $r =$ `introot`(p, k) vérifie donc toujours les inégalités :

$$r^k \leq p < (r+1)^k$$

Exemple

Comparaison avec `int`(w^(1/3)) qui dépend de la précision.

```
w=(10^20)^3
print introot(w,3);int(w^(1/3))
```

Sortie (200 ms)

```
100000000000000000000  99999999999991873536
```

INTLG(p)

V_fonction Partie entière du logarithme base 2

p

réel $p > 0$

Comme pour `intsqr` et `introot`, la précision de la fonction `intlg` est infinie (quel que soit `precision2`). L'entier $k =$ `intlg`(p) vérifie donc toujours les inégalités :

$$2^k \leq p < 2^{k+1}$$

Exemple

Le calcul par `int`(`log`(w,2)) ne donne pas `intlg`(w) par suite des erreurs d'arrondi.

```
w=2^100
print intlg(w);int(log(w,2))
```

Sortie (50 ms)

```
100  99
```

DIVR(p, q)

MODR(p, q)
MODS(p, q)
DIV(P, Q)
MOD(P, Q)

V_fonctions Division entière

p, q

réels ($q \neq 0$)

P, Q

réels exacts ($Q \neq 0$)

Lorsque p et q sont des nombres réels positifs, la division entière de p par q s'écrit $p = sq + r$ où s est un entier et où $r \in [0, q[$. Dans ce cas, s peut être obtenu par l'une quelconque des instructions :

```
s=int(p/q)
s=div(p,q)
s=divr(p,q)
s=p \ q
s=p div q
```

et r par :

```
r=mod(p,q)
r=modr(p,q)
r=p mod q
```

Exemple

Lorsque p et q ne sont pas positifs, les relations ci-dessus ne sont plus toutes équivalentes, comme l'illustre le programme suivant. Noter comment le caractère *f* (qui s'obtient par [a] S) est utilisé dans le `print` pour écrire des espaces.

```
print "   p  q  divr modr  div mod  mods"
forv q in (3,-3)
  forv p in (10,-10)
    print p;qff;divr(p,q);"  ";modr(p,q)ff;div(p,q)f;
     mod(p,q)ff;mods(p,q)
  nextv
nextv
```

Sortie (335 ms)

p	q	divr	modr	div	mod	mods
10	3	3	1	3	1	1
-10	3	-3	2	-4	2	-1
10	-3	-3	1	-3	1	1
-10	-3	3	2	4	2	-1

Nous donnons maintenant les définitions précises des diverses fonctions. Les nombres r_1 et r_2 équivalents à p modulo q tels que $r_1 \in [0, |q|[$ et $r_2 \in$

$[-|q|/2, |q|/2[$ sont renvoyés respectivement par `modr`(p, q) et `mods`(p, q). Le nombre r_1 peut également être obtenu par `mod`(p, q) et `p mod q`.

La fonction `div`(p, q) renvoie l'entier s_1 tel que $p = s_1 q + r_1$. La fonction `divr`(p, q) renvoie l'entier s_2 de même signe que pq tel que $|p| = |s_2 q| + r_2$ où $r_2 \in [0, |q|[$. Ce même entier s_2 est renvoyé aussi par les expressions :

```
p \ q
p div q
```

Les fonctions `mod` et `div` acceptent en réalité pour premier argument une expr exacte quelconque et pour deuxième argument un polynôme quelconque, et non seulement des réels exacts. Leur effet dans le cas général, ainsi que d'autres fonctions de division (en nombres complexes ou en polynômes modulaires) seront étudiées plus loin. Les fonctions `modr`, `mods` et `divr` n'acceptent comme arguments que des nombres réels, mais à la différence de `mod` et `div`, ces arguments peuvent être flottants. Dans le cas d'arguments flottants, `divr` renvoie une valeur exacte, `mods` et `modr` des valeurs flottantes.

Exemple

```
print modr(10,22/7);modr(10,pi)
print divr(10,22/7)ff;divr(10,pi)
```

Sortie (125 ms)

```
4/7  0.5752220392~
3    3
```

GCDR(e, f {, gi})

V_fonction Pgcd

e, f, gi

entiers

La fonction `gcdr` calcule le pgcd des entiers e, f, ...

Exemple

Calcule le pgcd de deux nombres d'une centaine de chiffres chacun. Le résultat peut être facilement vérifié puisque le pgcd de $2^p - 1$ et $2^q - 1$ est $2^r - 1$ où $r =$ `gcdr`(p, q).

```
print gcdr(2^370-1,2^430-1)
```

Sortie (30 ms)

```
1023
```

PRFACT(e [, f])

PRFACT$(e [, f])

V_ et C_fonctions Décomposition en facteurs premiers de e

e, f

entiers ($e \neq 0$, $e \neq 1$ par défaut $f = e$)

L'effet de la fonction `prfact` dépend de la valeur de f. Si $f = 0$, `prfact` cherche le plus petit facteur premier de $|e|$. Si $f > 0$, `prfact` cherche les facteurs premiers

de $|e|$ inférieurs à $f+1$. Si $f < 0$, `prfact` cherche le plus petit facteur premier de $|e|$, mais à la différence de $f = 0$, la recherche est limitée aux facteurs premiers inférieurs à $|f| + 1$. Ainsi, si f n'est pas donné, pour $e > 0$, `prfact` décompose complètement e en facteurs premiers. La factorisation obtenue est :

$$e = p_1^{a_1} \times p_2^{a_2} \times \cdots \times p_n^{a_n}$$

où $p_1 < p_2 < \cdots < p_n$ sont des nombres premiers, sauf peut-être p_n si f est donné. La fonction renvoie cette factorisation sous une forme non standard codée comme un polynôme illégal ($\oplus$ désigne la juxtaposition des monômes) :

$$P = p_1 u^{a_1} \oplus p_2 u^{a_2} \oplus \cdots \oplus p_n u^{a_n}$$

où u est le littéral `phantom` (de numéro zéro et de nom vide). Le polynôme P est illégal : les monômes sont ordonnés suivant la valeur des coefficients et non suivant les exposants, et des exposants identiques peuvent apparaître. Utiliser P dans des expressions n'a donc aucun sens, mais on peut l'afficher et le décortiquer. On obtient n par `polymn`(P), et en posant $M_i =$ `polym`(P, i), p_i et a_i sont donnés par `norm`(M_i) et `deg`(M_i).

La fonction `prfact$` effectue la même factorisation que `prfact`, mais écrit la décomposition sous une forme plus lisible en effectuant la transformation `change$` de l'exemple suivant. Elle convient si on désire seulement afficher le résultat.

Exemple

A l'écriture de P, le littéral `phantom` ne s'affiche pas. Le programme montre d'abord comment améliorer la sortie par `change$`, puis comment extraire les facteurs premiers et leurs exposants.

```
P=prfact(2^39+1)
print P
print change$(P,"*","","+","* ")
for i=1,polymn(P)
  M=polym(P,i)
  print norm(M);deg(M)
next i
```

Sortie (675 ms)

```
3*^2 +2731* +22366891*
3^2 * 2731 * 22366891
3  2
2731  1
22366891  1
```

Exemple

La factorisation de l'exemple précédent est effectuée par `prfact$`.

```
print 2^39+1;"=";prfact$(2^39+1)
```

Sortie (555 ms)

```
549755813889=  3^2 * 2731 * 22366891
```

Exemple

Le plus petit diviseur de e s'obtient par `norm(prfact(`e, 0`))`.

```
e=2^40+1
print e;norm(prfact(e,0))
```

Sortie (65 ms)

```
1099511627777  257
```

Performances de prfact

La fonction `prfact` fonctionne en divisant e par 2, 3, 5, 7, 11, puis par les entiers croissants exceptés les multiples de 2, 3, 5, 7 et 11. Elle convient pour trouver des facteurs inférieurs à 10^6, comme le montrent les temps de calculs de `prfact(`e, 0`)` en secondes pour obtenir le premier facteur g de e.

e	g	temps
$2^{31}-1$	premier	7
$2^{59}-1$	179951	39
$2^{107}+2^{54}+1$	843589	303

On trouvera dans la bibliothèque MATH des procédures mettant en œuvre des découvertes mathématiques récentes et permettant de factoriser des entiers jusqu'à 30 chiffres environ.

PHANTOM

Littéral interne de nom vide

C'est le littéral de numéro 0, utilisé dans `prfact`. Le littéral `phantom` peut être utilisé dans les expressions, mais il ne s'écrit pas en sortie.

PPWR(k)

V_fonction Factorielle de k

k

entier $k \in [0, 5909]$

La forme `ppwr(`k`)` renvoie la valeur exacte $k!$.

Exemple

Le programme suivant calcule plusieurs factorielles, et écrit le temps de calcul et le nombre d'octets occupés par le résultat. Il n'écrit pas les factorielles, ce qui prendrait plus de temps que le calcul lui même. Par exemple, l'écriture seule de factorielle 1000, qui utilise un écran et demi, prend 7 secondes (4 s pour la conversion du binaire en chaîne de caractère, et 3 s d'écriture), alors que le calcul de la factorielle est effectué en 5 s.

```
print "   k      ms    octets"
forv k in (10,100,[1000,5000,1000])
  clear timer
  w=ppwr(k)
```

```
        print justr$(k,4);justr$(mtimer,8);justr$(mlen(w),8)
      nextv
```

Sortie

```
    k       ms    octets
   10       20      12
  100      105      74
 1000     4810    1076
 2000    20155    2390
 3000    47505    3800
 4000    87770    5272
 5000   141645    6788
```

PPWR(p, k)

V_fonction Symbole de Pochhammer $p^{(k)}$

p

expr

k

entier

La fonction ppwr(p, k) renvoie 1 si $k = 0$. Si $k > 0$ elle renvoie le produit de k facteurs :

$$p^{(k)} = p(p-1)(p-2) \times \cdots \times (p-k+1).$$

Si $k < 0$ elle renvoie $1/(p-k)^{(-k)}$.

Cette fonction vérifie la relation `ppwr(p,k)=p*ppwr(p-1,k-1)`. Noter que l'argument p n'est pas limité aux seuls réels, mais peut être une expression quelconque.

Exemple

Le calcul suivant montre que le produit de 4 entiers consécutifs (a représente un entier) plus 1 est le carré d'un entier.

```
      print formf(ppwr(a,4));" + 1 =";formf(ppwr(a,4)+1)
```

Sortie (490 ms)

```
 [a]* [a -3]* [a -2]* [a -1] + 1 =   [a^2 -3*a +1]^2
```

Degré, Ordre et Coefficients

DEG(p {, xi})

ORD(p {, xi})

V_fonctions Degré et ordre (valuation)

p

poly

xi

littéral

Si aucun littéral n'est donné, les fonctions `deg`(p) et `ord`(p) donnent le degré et l'ordre total de p suivant tous ses littéraux. Le degré d'un monôme :

$$y^b \times \cdots \times x_1^{a_1} \times \cdots \times x_n^{a_n}$$

suivant les littéraux $x_1, \ldots, x_n$ est $a_1 + \cdots + a_n$. La fonction `ord`(p, x_1, ..., x_n) [resp `deg`(p, x_1, ..., x_n)] renvoie le minimum (resp le maximum) des degrés suivant les littéraux $x_1, \ldots, x_n$ des monômes de p. Toutefois si p ne contient aucun des littéraux $x_1, \ldots, x_n$, alors `ord`(p, x_1, ..., x_n) renvoie -1 au lieu de 0. Ainsi `ord`(p, y) vaut -1 si et seulement si p est un polynôme indépendant de y.

Le polynôme p (supposé non constant) est homogène si et seulement si `deg`(p)=`ord`(p). p est un polynôme homogène en X, Y (de degré $\neq 0$ en X, Y) si et seulement si `deg`(p, X, Y)=`ord`(p, X, Y).

Exemple

Le polynôme p est homogène en t, x.

```
      p=x^2*a + x*a^4*t
      print "Pour p=";p
      print1 a
      print1 t
      print1 x
      print1 y
      print2 a,t
      print2 a,x
      print2 t,x
      print "ord(p)    =";ord(p);"    deg(p)    =";deg(p)
      stop
print1:print "ord(p,@1)  =";ord(p,@1);"    deg(p,@1)  =";deg(p
 ,@1)
      return
print2:print "ord(p,@1,@2)=";ord(p,@1,@2);"    deg(p,@1,@2)=";
 deg(p,@1,@2)
      return
```

Sortie (440 ms)

```
Pour p=  x^2*a +x*a^4*t
ord(p,a)  =  1    deg(p,a)  =  4
ord(p,t)  =  0    deg(p,t)  =  1
ord(p,x)  =  1    deg(p,x)  =  2
```

```
ord(p,y)   = -1    deg(p,y)   =  0
ord(p,a,t)=  1     deg(p,a,t)=  5
ord(p,a,x)=  3     deg(p,a,x)=  5
ord(p,t,x)=  2     deg(p,t,x)=  2
ord(p)     =  3    deg(p)     =  6
```

DEGF(p, x)

ORDF(p, x)

V_fonction Degré et ordre en x

p

expr

x

littéral

Si p est de la forme $p = q \times A \times x^a$ où A est un polynôme d'ordre 0 en x (ord(A,x) ≤ 0), q est indépendant de x et a est un entier relatif, les fonctions ordf(p, x) et degf(p, x) renvoient respectivement a et $a +$ deg(A, x). Lorsque p est un polynôme dépendant de x, ces fonctions coïncident avec ord(p, x) et deg(p, x).

Exemple

Ordre et degré en x de l'expression :

$$p = \frac{a+b}{x^7} + \frac{7}{ax^5} + \frac{12x^{15}}{a+b+1}.$$

```
p=(a+b)/x^7+7/a/x^5+12*x^15/(a+b+1)
print  ordf(p,x);degf(p,x)
```

Sortie (130 ms)

```
 -7  15
```

Dans toute forme factorisée p le facteur x^a est apparent. La fonction ordf(p, x) fonctionne en réalité sans restriction sur p et renvoie l'exposant a. Par contre degf exige la forme particulière de p.

Exemple

La fonction degf sort en erreur pour :

$$p = \frac{1+x}{(1-x)x^{127}}.$$

```
p=(1+x)/(1-x)*x^-127
print  ordf(p,x);degf(p,x)
```

Sortie (55 ms)

```
 -127
*ERREUR* DEVELOPPEMENT EN X^-K
print  ordf(p,x);degf(p,x?
2.print  ordf(p,x);degf(p,x)
```

COEF(p {, xi, ki})

V_fonction Coefficient

p

poly

xi

littéral

ki

entier $k_i \in [0, 2^{16}[$

Les $2n$ arguments après p définissent le monôme normalisé $x_1^{k_1} \times x_2^{k_2} \times \cdots \times x_n^{k_n}$. Le dernier argument k_n peut être omis (par défaut $k_n = 1$). La fonction `coef` renvoie le polynôme $Q_{k_1,k_2,\ldots,k_n}$ indépendant des littéraux x_1, x_2, …, x_n tel que

$$p = \sum_{k_1,k_2,\ldots,k_n} Q_{k_1,k_2,\ldots,k_n} \times x_1^{k_1} \times x_2^{k_2} \times \cdots \times x_n^{k_n}.$$

Exemple

Détermination des coefficients de Y^k puis de Y^2Z du polynôme :

$$p = (5X + \frac{ZT}{11})Y^2 + 9Y + X + Z + U.$$

```
   p=(5*X+Z*T/11)*Y^2+9*Y+X+Z+U
   print "Pour p=";p
   t (p,Y,0)
   t (p,Y,1)
   t (p,Y,2)
   t (p,Y,3)
   t (p,Y,2,Z,1)
   stop
 t:print "coef@1=";tab(17),coef@1
   return
```

Sortie (320 ms)

```
Pour p=  5*X*Y^2 +X +1/11*Z*T*Y^2 +Z +9*Y +U
coef(p,Y,0)=       X +Z +U
coef(p,Y,1)=       9
coef(p,Y,2)=       5*X +1/11*Z*T
coef(p,Y,3)=       0
coef(p,Y,2,Z,1)=   1/11*T
```

SROOT(p [, x])

V_fonction Somme des racines

p

poly

x

littéral [par défaut $x =$ `polyl`(p)]

Posons $p = A_n x^n + A_{n-1} x^{n-1} + \cdots + A_0$ où les A_i $(i = 0, \ldots, n)$ sont indépendants de x et $n =$ `deg`(p, x) est le degré en x du polynôme p, La fonction `sroot` renvoie $-A_{n-1}/A_n$, qui est égal à la somme des racines en x de l'équation $p = 0$. Cette valeur peut aussi être calculée par :

```
-coef(p,x,n-1)/coef(p,x,n)
```

Lorsque p est du premier degré en x, la fonction renvoie la racine de l'équation $p = 0$.

Exemple

Le programme suivant résout le système d'équations :

$$\begin{cases} 3x + 7y = 17 \\ 8x - 2y = 19 \end{cases}$$

en éliminant x (puis y) entre les deux équations. Noter qu'on pouvait ici plus simplement utiliser `sroot`(w), sans préciser le littéral.

```
forv v in (x,y)
  w=elim(3*x+7*y-17,8*x-2*y-19,v)
  print v;"=";sroot(w,x+y-v)
nextv
```

Sortie (100 ms)

```
x=  79/62
y=  167/62
```

COEFF(p , x, [k])

V_fonction Coefficient

p

expr

x

littéral

k

entier*32 (par défaut $k = 1$)

Si $p = \sum_{k=a}^{b} Q_k \times x^k$ où Q_k est une forme factorisée indépendante du littéral x, `coeff`(p, x, k) renvoie Q_k. Les limites de la somme sont données par $a =$ `ordf`(p, x) et $b =$ `degf`(p, x). Si $k \notin [a, b]$ la fonction `coeff` renvoie 0. Dans le cas ou p est un polynôme les calculs sont plus rapides par `coef`, `deg` et `ord` que par `coeff`, `degf` et `ordf`.

Exemple

Le programme suivant calcule les éléments de matrice de r^k entre des fonctions d'onde de l'atome d'hydrogène. La fonction `esum`(f, b, r) qui renvoie l'intégrale $\int_0^\infty f(r) \exp(-br)\, dr$ utilise `coeff` pour obtenir le coefficient de r^i. Ici,

`coef` ne conviendrait pas, parce que $f(r)$ n'est pas un polynôme. Le littéral a représente le rayon de Bohr.

```
     for k=-2,5
       print "<2s|r^";justl$(k);"|2s>=";esum(1/(2*a^3)*r^2
        *(1-r/2/a)^2*r^k,1/a,r)
       print "<2p|r^";justl$(k);"|2p>=";esum(1/(24*a^5)*r^4
        *r^k,1/a,r)
     next k
     stop
esum:function(f,b,r)
     local index i
     for i=ordf(f,r),degf(f,r)
       vadd value,coeff(f,r,i)*ppwr(i)*b^-(i+1)
     next i
     return
```

Sortie (2730 ms)

```
<2s|r^-2|2s>=  1/4* [a]^-2
<2p|r^-2|2p>=  1/12* [a]^-2
...
<2s|r^5|2s>=  27720* [a]^5
<2p|r^5|2p>=  15120* [a]^5
```

Partie principale et Contenu

Les fonctions `red` et `redf` introduites ici permettent de simplifier des équations. Ainsi pour résoudre $p = 0$ en x, on remplacera d'abord p par `redf`(p, x), `red`(p, x) ou `red`(`redf`(p, x), x).

CONT(p { , xi })

V_fonction Contenu

RED(p { , xi })

V_fonction Partie principale

p

poly

xi

littéral

Les n arguments après p définissent n littéraux x_1, x_2, ..., x_n. Si aucun littéral n'est donné, on prend par défaut tous les littéraux de p. Soient A et B les deux

polynômes tels que $p = A \times B$ où A ne contient pas les littéraux x_1, x_2, ..., x_n, où B est un polynôme normalisé et où les polynômes `coef`(B, x_1, k_1, ..., x_n, k_n) (pour divers k_1, ..., k_n) ont pour pgcd 1.

Les fonctions `cont` et `red` renvoient respectivement sA et sB, où $s = \pm 1$. Lorsque aucun littéral n'est donné, pour les formes `red`(p) et `cont`(p), $s = 1$.

Exemple

Contenus et parties principales du polynôme $p = 17(x + y)(y - 3)/19$. La résolution de l'équation $p(x) = 0$ en x est immédiate en posant `red`(p, x)=0.

```
    p=17/19*x*y -51/19*x +17/19*y^2 -51/19*y
    pr cont(p,x)
    pr red(p,x)
    pr cont(p)
    pr red(p)
    stop
pr:print "@1=";tab(10),@1
    return
```

Sortie (240 ms)

```
cont(p,x)=  17/19*y -51/19
red(p,x)=   x +y
cont(p)=    17/19
red(p)=     x*y -3*x +y^2 -3*y
```

CONTF(p , x)

V_fonction Contenu

REDF(p , x)

V_fonction Partie principale

p

expr

x

littéral

Les facteurs du codage interne de p (p est transformé en forme factorisée si besoin est) sont groupés en $p = q \times r \times x^a$ où q est indépendant du littéral x et où r est un produit de polynômes normalisés dont chacun dépend de x. La fonction `contf`(p, x) renvoie q, et si r est un polynôme, la fonction `redf`(p, x) renvoie r. On notera que, en général, r peut encore être réduit par `red`(r, x) comme dans l'exemple suivant.

Exemple

La fonction `redf` renvoie le facteur r de p, et `red` en extrait le facteur $x + 4$.

```
r=(z+3)*(x+4)
q=125/(z-2)
p=q*r*x^-5
```

```
        print "p=";p
        print contf(p,x)
        r1=redf(p,x)
        print r1a;cont(r1,x)a;red(r1,x)
```

Sortie (245 ms)

```
  p=  125* [x]^-5* [z -2]^-1* [z*x +4*z +3*x +12]
    125* [z -2]^-1
    z*x +4*z +3*x +12
    z +3
    x +4
```

Substituer et Homogénéiser

SUBS(p { , xi = qi })

SUBSR(p { , Mi = qi })

SUBSRR(p { , Mi = qi })

V_fonctions Substitutions

p, qi

expr

xi

littéral

Mi

monôme normalisé

La fonction `subs`(p, $x_1 = q_1$, $x_2 = q_2$, ...) renvoie p en remplaçant le littéral x_1 par q_1, le littéral x_2 par q_2, ... La fonction `subsr` utilise n monômes de la forme $M_i = x_1^{k_1} \times x_2^{k_2} \times \cdots \times x_p^{k_p}$. Elle transforme p suivant le procédé suivant :
Pour $i = 1, 2, \ldots, n$
a) Récrire $p = N/D$ où $N =$ `num`(p) et $D =$ `den`(p) sont des polynômes.
b) Faire apparaître autant que possible le monôme M_i.
c) Remplacer partout en même temps dans p, M_i par q_i.

Si on utilise pour M_i le littéral x_i, `subsr`(p, $x_1 = q_1$, $x_2 = q_2$, ...) donne le même résultat que `subs`(p, $x_1 = q_1$, $x_2 = q_2$, ...), mais lorsque p est factorisé, `subs` qui ne développe pas les facteurs est plus rapide. Par exemple :

```
W=subsr((x+1)^100,x=y+1)
```

prend 6 s, alors qu'avec `subs`, en mode `factor`, le calcul prend 0.02 s.

La fonction `subsrr` effectue les mêmes transformations que `subsr`, puis recommence ces mêmes transformations jusqu'à ce que p reste inchangé pendant tout un cycle de transformations. Cela peut donner une boucle sans fin comme dans :

```
W=subsrr(x+1,x=y,y=x)
```

Exemple

Noter que `subs` renvoie une forme factorisée. La deuxième instruction montre l'égalité $(1+\sqrt{2})^8 = 577 + 408\sqrt{2}$.

```
print subs((x+y)^2,x=x^5+1,y=-1)
print subsr((1+s)^8,s^2=2)
print subsr(x^16,x^2=x)
print subsrr(x^16,x^2=x)
print subsrr(a*x^4+b*x^2+c,x^2=y*x)
```

Sortie (375 ms)

```
 [x]^10
408*s +577
x^8
x
x*y^3*a +x*y*b +c
```

HOMOG(p, x)

V_fonction Homogénéisation

p

expr ne contenant pas x

x

littéral

La fonction `homog` homogénéise p par le littéral x.

Exemple

L'exemple montre aussi comment déshomogénéiser par `subs`.

```
W=homog( (1+3*z+7*z^2)/(9+a^14),x)
print W
print subs(W,x=1)
```

Sortie (180 ms)

```
[a^14 +9*x^14]^-1* [7*z^2 +3*z*x +x^2]
[a^14 +9]^-1* [7*z^2 +3*z +1]
```

Dérivation, Intégration et Partie singulière

DER(p {, x})
DERM(p , k [, x])

V_fonctions Dérivation

p

expr

x

littéral

k

entier $k \geq 0$

La fonction der(p, x_1, x_2, ..., x_n) dérive p suivant x_1, puis suivant x_2, ..., x_n. La fonction derm(p, k, x) dérive p, k fois suivant x. Si aucun littéral n'est donné dans ces fonctions, la dérivée est calculée par rapport à polyl(p) (ou polyl(factorp(p, 2)) si p est factorisé).

Exemple

Le programme dérive $(ax + b)/(cx + b)$ par rapport à x, puis diverses dérivées de $ax^3 + px^2 + qx$.

```
print der((a*x+b)/(c*x+d),x)
w=a*x^3+p*x^2+q*x
print der(w,x,x,a)
for i=0,4
  print i;derm(w,i,x)
next
```

Sortie (340 ms)

```
 [x*c +d]^-2* [a*d -b*c]
6*x
0  a*x^3 +x^2*p +x*q
1  3*a*x^2 +2*x*p +q
2  6*a*x +2*p
3  6*a
4  0
```

Exemple

Le programme montre que l'équation de la tangente au point $x = a$ de la parabole $y = f(x) = 2x^2 + 3$ est :

$$y - 4ax + 2a^2 - 3 = 0.$$

La commande lit fixe l'ordre des littéraux pour améliorer la sortie.

```
lit y
f=2*x^2+3
w=(y-subs(f,x=a))-subs(der(f,x),x=a)*(x-a)
print w
```

Sortie (110 ms)

```
y -4*x*a +2*a^2 -3
```

INTG(p {, x})

INTGM(p , k [, x])

V_fonctions Intégration

p

expr

x

littéral

k

entier $k \geq 0$

La fonction `intg`(p, x_1, x_2, ..., x_n) intègre p suivant x_1, puis suivant x_2, ..., x_n. La fonction `intgm`(p, k, x) intègre p, k fois suivant x. Si aucun littéral n'est donné dans ces fonctions, l'intégrale est calculée par rapport à `polyl`(p) (ou `polyl`(`factorp`(p, 2)) si p est factorisé).

Exemple

Le programme détermine l'aire s limitée par la parabole $y = f(x) = (b-x)(a-x)$ et l'axe des x :

$$s = \int_a^b f(x)\,dx.$$

Le résultat montre que $s = (b-a)^3/6$.

```
f=(b-x)*(x-a)
w=intg(f,x)
s=subs(w,x=b)-subs(w,x=a)
print formf(s)
```

Sortie (265 ms)

```
1/6* [b -a]^3
```

Exemple

La première intégrale donne le résultat :

$$\int \frac{4x^3+21x^2+36x+21}{x^6+12x^5+58x^4+144x^3+193x^2+132x+36}dx = \frac{-(2x+3)}{(x+3)(x+2)(x+1)}.$$

```
print intg((4*x^3 +21*x^2 +36*x +21)/(x^6 +12*x^5 +58*x
 ^4 +144*x^3 +193*x^2 +132*x +36))
print intgm(a,10,x)
print intg(1/(x^2-3*x+2),x)
```

Sortie (1840 ms)

```
- [2*x +3]* [x +3]^-1* [x +2]^-1* [x +1]^-1
 1/3628800* [a]* [x]^10
```

```
[£og(x-2) -£og(x-1)]
```

Comme expliqué ci-dessous, les performances de `intg` sont limitées. On trouvera cependant dans la bibliothèque MATH des procédures permettant d'intégrer toutes les fractions rationnelles sans restriction. La fonction `intg` a besoin des pôles de p. Si elle ne peut pas les trouver il y a erreur Intégration.

Exemple

Les pôles sont non rationnels et donc introuvables par `intg`.

```
print intg(1/(X^2+X+1))
```

Sortie (160 ms)

```
*ERREUR* INTEGRATION
print intg(1/(X^2+X+1)?
```

En cas de pôle de résidu $\neq 0$ en a, l'intégrale contient $\log(|x-a|)$. Cela apparaît sous la forme `£og(x-a)` dans le résultat, mais attention cette forme n'est pas une fonction de x, mais le nom d'un littéral créé par Basic 1000d. On ne peut donc ni le dériver ni l'intégrer de façon correcte.

Exemple

En principe $W = \log|X-1|$, cependant le calcul de la dérivée de W par `der` donne 0, et son intégration par `intg` est fausse.

```
W=intg(1/(X-1))
print W;der(W, X)
print intg(W,X)
```

Sortie (165 ms)

```
£og(X-1)  0
 [£og(X-1)]* [X]
```

Si £og(x-0) apparaît plusieurs fois, c'est en fait un nouveau littéral chaque fois.

Exemple

La sortie n'est pas nulle pour le Basic.

```
print intg(1/x)-intg(1/x)
```

Sortie (175 ms)

```
[£og(x-0) -£og(x-0)]
```

Il est donc clair que si le résultat d'une intégration multiple par `intgm` contient des `£og(x...)`, l'intégrale peut être fausse, car ces `£og(x...)` ont été considérés comme des constantes dans les intégrations successives.

PSING(p, x, q, y)

V_fonction Partie singulière

p, q

expr

x, y

littéraux (p et q ne doivent pas contenir le littéral y)

Si

$$p = p_0 + \sum_{i=1}^{m} \frac{A_i}{(x-q)^i}$$

où p_0 est régulier en $x = q$ et A_i est indépendant de x, la fonction `psing` renvoie la partie singulière de p, mise sous la forme

$$\sum_{i=1}^{m} \frac{A_i}{y^i}.$$

Exemple

Le programme montre que la décomposition en éléments simples de la fraction rationnelle p est donnée par :

$$p = x^2 + x + \frac{1}{x-1} + \frac{1}{(x-1)^2} + \frac{1}{(x-1)^3}$$

```
p=(x^5 -2*x^4 +3*x^2 -2*x +1)/(x^3 -3*x^2 +3*x -1)
q=psing(p,x,1,y)
print str$(q,y)
print formd(p-subs(q,y=x-1))
```

Sortie (385 ms)

```
( 1)*y^-1+( 1)*y^-2+( 1)*y^-3
 x^2 +x
```

Développements limités

TAYLOR(p , k [, x])

V_fonction Développement limité

p

expr

k

entier $k \geq 0$

x

littéral

La fonction `taylor`(p, k, x) renvoie :

$$Q = F \times x^a \times (C_0 + C_1 x + C_2 x^2 + \cdots + C_k x^k)$$

où F est une forme factorisée indépendante de x, C_i pour $i = 0, \ldots, k$ est un polynôme indépendant de x et a est un entier. Cette expression Q est le développement limité de p à l'ordre $a+k$ en x au voisinage de $x = 0$. Si $a \geq 0$, Q est le développement de Mac-Laurin à l'ordre $a+k$. Lorsque x est omis, la fonction utilise par défaut $x =$ `polyl`(p) (ou $x =$ `polyl(factorp`(p, 2)) si p est factorisé)

Exemple

```
print taylor(x^2/(1-x),5)
```

Sortie (110 ms)

```
[x]^2* [x^5 +x^4 +x^3 +x^2 +x +1]
```

Exercice Boulanger

On dispose de pièces de 1, 5 et 10 francs. De combien de façons différentes peut on régler la note (N francs) de son boulanger? Par exemple si la note est de 50 francs, une façon possible est avec 10 pièces de 5 F, une autre façon est avec 4 pièces de 10 F et 2 de 5 F, etc. Donner la réponse pour tous les N de 1 à 50.

SATN(p , k [, x])

SSIN(p , k [, x])

SCOS(p , k [, x])

SEXP(p , k [, x])

SLOG1(p , k [, x])

SHYG(p , k , x {, qi , mi })

p

poly `ord`(p, x)> 0

k

entier $k \geq 0$

x

littéral [par défaut $x =$ `polyl`(p)]

qi

expr

mi

entier*16

La fonction `satn` renvoie le développement limité de arctan p à l'ordre k en x (au voisinage de 0). Ce résultat est obtenu en supprimant tous les termes x^n de degré $n > k$ dans la série :

$$\arctan p = p - \frac{p^3}{3} + \frac{p^5}{5} - \cdots$$

Les fonctions `ssin`, `scos`, `sexp` et `slog1` sont analogues à `satn` pour les séries :

$$\begin{aligned}\sin p &= p - p^3/3! + p^5/5! - \cdots \\ \cos p &= 1 - p^2/2! + p^4/4! - \cdots \\ \exp p &= 1 + p + p^2/2! + p^3/3! + \cdots \\ \log(1+p) &= p - p^2/2 + p^3/3 - \cdots\end{aligned}$$

La forme `shyg`(p, k, x, q_1, m_1, ..., q_n, m_n) correspond à la fonction hypergéométrique généralisée :

$$\begin{aligned}&1 + q_1^{m_1} \ldots q_n^{m_n} p + q_1^{m_1}(q_1+1)^{m_1} \ldots q_n^{m_n}(q_n+1)^{m_n} p^2 \\ &\quad + q_1^{m_1}(q_1+1)^{m_1}(q_1+2)^{m_1} \ldots q_n^{m_n}(q_n+1)^{m_n}(q_n+2)^{m_n} p^3 + \cdots\end{aligned}$$

La fonction hypergéométrique :

$$F(a,b,c,x) = 1 + \frac{ab}{c}\frac{x}{1!} + \frac{a(a+1)b(b+1)}{c(c+1)}\frac{x^2}{2!} + \cdots$$

s'obtient à l'ordre k par `shyg(x,k,x,a,1,b,1,c,-1,1,-1)`.
La fonction hypergéométrique dégénérée :

$$F(a,c,x) = 1 + \frac{a}{c}\frac{x}{1!} + \frac{a(a+1)}{c(c+1)}\frac{x^2}{2!} + \cdots$$

s'obtient à l'ordre k par `shyg(x,k,x,a,1,c,-1,1,-1)`.

Exemple

Développement à l'ordre 7 de cotg x

```
print "cotg(x)=";str$(taylor(scos(x,9)/ssin(x,9),9),/x)
 ;" + ..."
```

Sortie (640 ms)

```
cotg(x)= (  1)*x^-1+( -1/3)*x+( -1/45)*x^3+( -2/945)*x^5
 +( -1/4725)*x^7 + ...
```

Exemple

Calcul du développement à l'ordre 12 de exp x, qui est utilisé pour calculer une valeur approchée de la base des logarithmes naturels e.

```
factor
S=sexp(x,12)
print fsubs(S,x=1~)
```

Sortie (595 ms)

```
0.2718281828~ E+1
```

Exemple

La fonction `shyg` permet le développement en série de nombreuses fonctions usuelles. La série

$$(1+x)^s = 1 + \frac{(-s)(-x)}{1} + \frac{(-s)(-s+1)(-x)^2}{2!} + \cdots$$

est ici calculée à l'ordre 5.

```
w=shyg(-x,5,x,-s,1,1,-1)
print "(1+x)^s=";str$(w,/x);"+..."
```

Sortie (740 ms)

```
(1+x)^s= (  1)+(   [s])*x+(  1/2* [s]* [s -1])*x^2+(  1/6* [s]* [s^2 -
 3*s +2])*x^3+(  1/24* [s]* [s^3 -6*s^2 +11*s -6])*x^4+(  1/120* [s]*
 [s^4 -10*s^3 +35*s^2 -50*s +24])*x^5+...
```

Déterminant, Elimination et Racines

DET(D, n [, k])

V_fonction Déterminant

n

entier $n \in [0, 999]$

k

entier $k \in [0, n]$ (par défaut $k = 1$)

D

nom

Le nom `D` est tel que `D`(i, j) désigne une expression pour i et j entiers $\in [k, n]$. Le nom `D` peut être un tableau déclaré par `var D(M, N)`, (ou `index D(M, N)`) ou une fonction à 2 arguments. Chaque `D`(i, j), qui doit représenter la valeur de l'élément $D_{i,j}$ du déterminant, est appelé une fois et une seule par la fonction `det`, et mémorisé dans la pile des variables internes. L'ordre d'appel est $D_{k,k}$, $D_{k,k+1}$, ..., $D_{k+1,k}$, ..., $D_{n,n}$. La variable d'état `s_var` doit donc être plus grande que le nombre d'éléments du déterminant, $(n - k + 1)^2$.

La fonction `det` calcule le déterminant :

$$\begin{vmatrix} D_{k,k} & \dots & D_{k,n} \\ \vdots & \ddots & \vdots \\ D_{n,k} & \dots & D_{n,n} \end{vmatrix}$$

Si les $D_{i,j}$ sont tous des nombres flottants (réels ou complexes), le calcul est effectué de façon approchée. Si les $D_{i,j}$ sont tous des expressions exactes (polynômes ou formes factorisées), le déterminant est calculé exactement. La fonction `det` utilise la méthode de Bareiss et a un temps de calcul polynomial en n. Pour des $D_{i,j}$ polynomiaux, `det` est plus rapide avec l'option `develop`.

Exemple

Le déterminant à circulation :

$$\begin{vmatrix} a & b & c \\ c & a & b \\ b & c & a \end{vmatrix}$$

est calculé par `det`. On vérifie que sa valeur est égale à :

$$(a+b+c)(a+bj+cj^2)(a+bj^2+cj)$$

où j est une racine de $j^2+j+1=0$ à l'aide d'un calcul qui impose cette relation en j par `cond`.

```
var D(3,3)
for m=1,3
  for n=1,3
    read D(m,n)
next n,m
data a,b,c,c,a,b,b,c,a
w=det(D,3)
print w
cond j^2+j+1
W=(a+b+c)*(a+b*j+c*j^2)*(a+b*j^2+c*j)
print W
```

Sortie (320 ms)

```
a^3 -3*a*b*c +b^3 +c^3
a^3 -3*a*b*c +b^3 +c^3
```

Exemple

Le calcul de l'exemple précédent est effectué à l'aide de la fonction `D` au lieu d'un tableau `D`.

```
  var d(2)
  read d(0),d(1),d(2)
  data a,b,c
  w=det(D,3)
  print w
  stop
D:function(i,j)
  value=d(modr(j-i,3))
  return
```

Sortie (220 ms)

```
a^3 -3*a*b*c +b^3 +c^3
```

ELIM(p , q [, x])

V_fonction Elimination de x entre p et q

p, q

poly

x

littéral [par défaut $x = \texttt{polyl}(q)$]

Si :

$$p = A_0 x^n + A_1 x^{n-1} + \cdots + A_n$$
$$q = B_0 x^m + B_1 x^{m-1} + \cdots + B_m$$

où A_i et B_j sont indépendants de x, la fonction `elim` calcule la résultante de p et q, donnée par le déterminant d'ordre $m+n$:

$$\begin{vmatrix} A_0 & A_1 & A_2 & \dots & A_n & 0 & \dots & 0 \\ 0 & A_0 & A_1 & \dots & A_{n-1} & A_n & \dots & 0 \\ & & \ddots & \ddots & \ddots & \ddots & \ddots & \\ 0 & 0 & \dots & A_0 & A_1 & A_2 & \dots & A_n \\ B_0 & B_1 & \dots & B_m & 0 & 0 & \dots & 0 \\ 0 & B_0 & \dots & B_{m-1} & B_m & 0 & \dots & 0 \\ & & \ddots & \ddots & \ddots & \ddots & & \\ 0 & 0 & \dots & 0 & B_0 & \dots & B_{m-1} & B_m \end{vmatrix}$$

Si p et q ne contiennent pas x `elim` renvoie -1.

Exemple

La résolution du système d'équations :

$$\begin{cases} x + y + z = a + 3 \\ xyz = 2a \\ x^2 + y^2 + z^2 = a^2 + 5 \end{cases}$$

est obtenue en éliminant y et z. Les racines en x sont les zéros de W, et se lisent de suite (1, 2 et a) sur la forme factorisée.

```
W1=elim( x+y+z-a-3, x*y*z-2*a , y)
W2=elim( x+y+z-a-3, x^2+y^2+z^2-a^2-5, y)
W=elim(W1, W2, z)
print formf(W)
```

Sortie (520 ms)

```
4* [x -2]^2* [x -1]^2* [x -a]^2
```

Exemple

Avant l'élimination de x entre p et q, il vaut mieux pour la rapidité de `elim` remplacer p et q par leurs parties principales `red`(p, x) et `red`(q, x).

```
p=(x^2-y)*a
q=(b-1)*(x+c)
```

```
print elim(p,q,x)
print elim(red(p,x),red(q,x),x)
```

Sortie (200 ms)

```
-y*a*b^2 +2*y*a*b -y*a +a*b^2*c^2 -2*a*b*c^2 +a*c^2
 y -c^2
```

ROOT(p , k)

V_fonction Racine k-ième exacte

p

expr

k

entier $k > 0$

S'il existe une expression rationnelle q telle que $p = q^k$, la fonction `root` renvoie q. Sinon elle renvoie 0.

Exemple

```
print root((A^2+2*A*B+B^2)/C^2, 2)
print root(3,2)
```

Sortie (95 ms)

```
  [C]^-1* [A +B]
 0
```

Division et PGCD

DCOM(p1 , p2 { , pj })

V_fonction Diviseur commun

p1, p2, pj

expr

La fonction `dcom`(p_1, ..., p_n) renvoie une expression q telle que p_1/q, ..., p_n/q soient des polynômes de pgcd égal à une constante. La valeur q est sous forme produit avec un facteur constant égal à 1.

Exemple

En utilisant en entrée de `dcom` les produits des polynômes $x-1$, $y+1$ et $(x-1)(y+1)$ par une fraction rationnelle q, on obtient en sortie q, à un facteur constant près.

```
q=17/23*(x+4)/(y+1)
p1=(x-1)*q
p2=(y+1)*q
```

```
        p3=(x-1)*(y+1)*q
        print dcom(p1,p2,p3)
```

Sortie (210 ms)

```
   [y +1]^-1* [x +4]
```

DIV(p , q [, x])

MOD(p , q [, x])

V_fonctions Division

p

expr

q

poly $q \neq 0$

x

littéral [par défaut $x =$ `polyl`(q)]

La valeur renvoyée par `mod` représente le reste de la division de p par q et s'obtient de la façon suivante :

Si q est réel

p doit être un poly et l'argument x est inutile. On remplace tous les coefficients C des monômes de p par `modr`$(C, |q|)$. En particulier si p et q sont réels positifs, `mod`(p, q)=`modr`(p, q)

Exemple

Calcul du reste modulo 3 de $\frac{17}{3}xy + 28z$.

```
        p=17/3*x*y+28*z
        print mod(p,3)
```

Sortie (50 ms)

```
  8/3*x*y +z
```

Si q est un poly non constant

• Si p est un poly, la division euclidienne de p par q s'écrit :

$$p = Q \times q + R$$

où Q et R sont des polynômes en x, mais pas forcément suivant les autres littéraux, qui vérifient `degf`$(R, x) <$ `deg`(q, x). Dans ce cas `mod`(p, q, x) renvoie R (sous forme factorisée).

• Si p est sous forme factorisée :

$$p = C_1 \times f_2^{e_2} \times \cdots \times f_n^{e_n}$$

chaque facteur $f_2, \ldots, f_n$ est remplacé par son reste $R_i =$ `mod`(f_i, q, x) calculé comme dans le cas p poly. Dans ce cas `mod`(p, q, x) renvoie :

$$C_1 \times R_2^{e_2} \times \cdots \times R_n^{e_n}.$$

La fonction div(p, q, x) renvoie

$$\frac{p - \mathrm{mod}(p, q, x)}{q}.$$

Exemple

Le premier calcul est effectué avec W factorisé et le deuxième avec une forme polynomiale.

```
factor
W=(X+1)*(X-1)
print mod(W,X^2+1,X)
print mod(formd(W),X^2+1)
print mod(formd(W),A*X^2+1)
```

Sortie (170 ms)

```
    [X -1]* [X +1]
 -2
 - [A]^-1* [A +1]
```

DIVD(p , q [, x])
DIVN(p , q [, x])
MODD(p , q [, x])
MODN(p , q [, x])

V_fonctions Pseudo-division

p, q

poly $q \neq 0$

x

littéral [par défaut $x =$ polyl(q)]

La pseudo-division euclidienne de p par $q = Bx^m + \cdots$ où $m =$ deg(q, x) et $B =$ coef(q, x, m) s'écrit :

$$B^a \times p = S \times q + R$$

où l'entier a est donné par $a =$ max(deg(p, x)$-m+1$, 0), S et R sont des polynômes et deg(R, x) $< m$. Le polynôme S est renvoyé par divn, le polynôme R est renvoyé par modn et le polynôme B^a est renvoyé par divd et modd (qui sont identiques).

Pour des polynômes p et q compliqués, il est conseillé d'utiliser ces fonctions plutôt que div et mod, qui nécessitent beaucoup plus de mémoire et de temps.

Exemple

La pseudo-division de p par q est ici plus rapide que la division par un facteur $2\frac{1}{2}$. Le résultat 0 correspond à la vérification de $B^a p = Sq + R$.

```
p=27*x^3+3
```

```
q=(a-1)*x-5
clear timer
S=divn(p,q,x)
Ba=divd(p,q,x)
R=modn(p,q,x)
print "pseudo-division";mtimer
print S
print Ba
print R
print Ba*p-S*q-R
clear timer
R=mod(p,q,x)
S=div(p,q,x)
print "division";mtimer
print S
print R
```

Sortie (620 ms)

```
pseudo-division  95
  27*x^2*a^2 -54*x^2*a +27*x^2 +135*x*a -135*x +675
  a^3 -3*a^2 +3*a -1
  3*a^3 -9*a^2 +9*a +3372
  0
division  240
  27* [a -1]^-3* [x^2*a^2 -2*x^2*a +x^2 +5*x*a -5*x +25]
  3* [a -1]^-3* [a^3 -3*a^2 +3*a +1124]
```

DIVE(A, B)

DIVEZ(A, B)

V_fonctions Division exacte

A, B

poly

Posons $R = A/B$. La fonction `dive` renvoie R si R est un polynôme. Sinon elle renvoie 0. La fonction `divez` renvoie R si R est un polynôme à coefficients entiers. Sinon elle renvoie 0.

Ces fonctions sont seulement 2 à 20 fois plus lentes que les multiplications de polynômes et beaucoup (jusqu'à 1000 fois) plus rapides que la division `A/B`. Elles ne doivent donc être nullement négligées.

Exemple

Comme $p = (f(a) - f(b))r$ est divisible par $q = (a - b)r$, on peut calculer p/q par `dive`(p, q). On compare les temps de la division par `dive`, `divn`, `div` et /. On compare aussi avec le temps de la multiplication $p * q$.

```
r=(7*y*z+y-1)
f=r*(1+x*y-3*x)^5
```

```
p=subs(f,x=a)-subs(f,x=b)
q=(a-b)*r
clear timer
U=dive(p,q)
print "dive";justr$(mtimer,7);" ms"
clear timer
U=divn(p,q)
print "divn";justr$(mtimer,7);" ms"
clear timer
U=div(p,q)
print " div";justr$(mtimer,7);" ms"
clear timer
U=p/q
print "   /";justr$(mtimer,7);" ms"
clear timer
U=p*q
print "   *";justr$(mtimer,7);" ms"
```

Sortie

```
dive    645 ms
divn  10690 ms
 div  17645 ms
   /   6915 ms
   *    330 ms
```

GCD(p1 , p2 { , pk })

V_fonction pgcd (polynômes)

p1, p2, pk

poly

La fonction `gcd`(p_1, ..., p_n) renvoie le pgcd des polynômes p_1, ..., p_n. Le pgcd est déterminé dans l'anneau $\mathbf{Z}[x, y, \ldots]$ (x, y, ... étant les littéraux des polynômes) ou $\mathbf{Q}[x, y, \ldots]$ suivant les coefficients des polynômes. Le calcul du pgcd est plus rapide dans le premier cas, ainsi `gcd`(`red`(p_1), `red`(p_2)) [calculs en entiers] est en général plus rapide que `gcd`(p_1, p_2) [s'il est calculé en fractions]. Cette fonction est très optimisée dans le premier cas (méthode des subrésultantes légèrement modifiée) car elle est appelée implicitement dans les calculs sous forme factorisée (pour réduire à des facteurs premiers deux à deux). La forme `gcd`(p, 0) est admise et vaut p.

Exemple

Comme les coefficients des polynômes p et q ont de grands dénominateurs, le calcul de `gcd`(p, q) demande 25 fois plus de temps que celui de `gcd`(`red`(p), `red`(q)).

```
r=(2^51+1)/(2^43+1)*(x-5)*(y+x)
p=r*(x-7)^15*(2^51+1)/(2^43+1)
```

```
q=r*(x-6)^11*(3^51+1)/(3^43+1)
clear timer
r=gcd(p,q)
print mtimer;
clear timer
r=gcd(red(p),red(q))
print mtimer
print r
```

Sortie (44 s)

```
 39675  1540
-x^2 -x*y +5*x +5*y
```

INV(A, B [, x])

V_fonction Inverse de A modulo B

A, B

poly

x

littéral [par défaut $x =$ `polyl`(B)]

Soit P le pgcd de A et B calculé par `gcd`$(A,\ B)$. La fonction `inv` renvoie le polynôme généralisé en x, U, tel que $U \times A = P$ modulo B et `degf`$(U,\ x) <$ `deg`$(B,\ x)$. En général U n'est pas un polynôme suivant les autres littéraux.

Exemple

L'identité suivante (Identité de Bezout) est vérifiée :

$$A \times U + B \times V = P$$

```
A=17*(x-3)*(x^4+1)
B=17*(x-3)*(3*x^3-5)
P=gcd(A,B)
U=inv(A,B)
V=inv(B,A)
print P
print U
print V
print A*U+B*V-P
```

Sortie (415 ms)

```
-17*x +51
-225/706*x^2 +135/706*x -81/706
 75/706*x^3 -45/706*x^2 +27/706*x +125/706
 0
```

Exemple

La fonction `inv` permet de travailler avec les nombres algébriques. Supposons que a et b soient des nombres algébriques définis par les équations :

$$\begin{cases} a^2 + a + 1 = 0 \\ b^2 - b - a = 0 \end{cases}$$

Toute expression rationnelle en a et b peut se simplifier en un polynôme de degré 1 en a ou b. Proposons nous le problème de simplifier ainsi l'inverse de $b + a + 1$. Le calcul suivant :

```
print inv(b+a+1,b^2-b-a,b)
```

Sortie (125 ms)

```
- [a^2 +2*a +2]^-1* [b -a -2]
```

montre que :

$$\frac{1}{b+a+1} = -\frac{b-a-2}{a^2+2a+2}$$

On montre ensuite que l'inverse de $a^2 + 2a + 2$ est $-a$ par :

```
print inv(a^2 +2*a +2,a^2+a+1)
```

Sortie (55 ms)

```
- [a]
```

Nous regroupons ces résultats, en simplifiant à l'aide de `mod`.

```
z=mod([a] * [b -a -2],a^2+a+1)
print z
```

Sortie (40 ms)

```
a*b -a +1
```

Nous avons ainsi montré que les nombres $ab - a + 1$ et $b + a + 1$ sont inverses. On peut le vérifier en simplifiant leur produit.

```
z=mod((a*b -a +1)*(b+a+1),b^2-b-a,b)
z=mod(z,a^2+a+1)
print z
```

Sortie (55 ms)

```
1
```

Décorticage

POLYN(p)

NORM(q)

V_fonctions Premier coefficient numérique

p

poly

q

expr

Le codage de la forme polynomiale de p s'écrit :

$$p = \sum_{i=0}^{m-1} c_i \prod_{j=1}^{n} x_j^{a_{i,j}}.$$

La fonction `polyn` renvoie le coefficient c_0 du premier monôme de ce codage. C'est aussi le premier nombre qui s'affiche par :

```
print formd(p)
```

La fonction `norm`(q) dépend du codage mémoire de q. Si ce codage est la forme factorisée :

$$q = P_1 \prod_{i=2}^{s} P_i^{k_i}$$

`norm` renvoie le nombre P_1 = `factorp`(q). Si le codage de $q = p$ est la forme polynomiale ci-dessus, `norm`(q) renvoie le nombre c_0 = `polyn`(q). Si p est un monôme, `norm`(p)=`polyn`(p) que la forme soit factorisée ou non.

Exemple

Si p est sous forme polynôme, on a `norm`(p)=`polyn`(p), par contre si p est sous forme factorisée en général `norm`(p)≠`polyn`(p).

```
factor
p=17/19*(2*x^5*y+1)
print formd(p)
print polyn(p)
print norm(p)
print norm(formd(p))
```

Sortie (100 ms)

```
34/19*x^5*y +17/19
34/19
17/19
34/19
```

POLYLN(p)

POLYL(p [, k])

V_fonctions Littéraux

p

poly

k

entier (par défaut $k = 1$)

La fonction `polyln`(p) renvoie le nombre de littéraux n de p. Ces n littéraux sont ordonnés suivant l'ordre de création (voir `lit`). Si $k \in [1, n]$ la fonction `polyl`(p, k) renvoie le k-ième littéral de p. Si $k \notin [1, n]$ elle renvoie 0.

Exemple

Les littéraux sont ici ordonnés ($X < Y < Z$) suivant l'ordre de décodage de l'expression W, mais si on fait précéder le programme de `lit Z,Y` cet ordre devient $Z < Y < X$.

```
W=5*X*Y+Z/6
print polyln(1);polyln(W)
for i=1,5
  print polyl(W,i);
next i
```

Sortie (105 ms)

```
0  3
X  Y  Z  0  0
```

Exemple

Le polynôme w est développé sous la forme :

$$w = A_m \times z^m + A_{m-1} \times z^{m-1} + \cdots + A_n \times z^n$$

et on montre comment obtenir les divers éléments (m, n, A_i) de cette décomposition pour les deux choix possibles a et x du littéral z.

```
w=a^3*x +5*a^2 +10/3*a*x +5/9*x^2
for ia=1,polyln(w)
  z=polyl(w,ia)
  print "Analyse en";z
  print " deg  coefficient"
  for ib=ord(w,z),deg(w,z)
    print ibf;coef(w,z,ib)
  next ib
next ia
```

Sortie (315 ms)

```
 Analyse en  a
  deg  coefficient
   0   5/9*x^2
   1   10/3*x
   2   5
   3   x
 Analyse en  x
  deg  coefficient
   0   5*a^2
   1   a^3 +10/3*a
```

```
  2   5/9
```

POLYMN(p)

POLYM(p [, k])

V_fonctions Monômes

p

poly

k

entier (par défaut $k = 1$)

La fonction `polymn`(p) renvoie le nombre de monômes m de p. Ces m monômes sont ordonnés suivant l'ordre lexicographique décroissant des exposants. C'est aussi l'ordre affiché par :

```
print formd(p)
```

Si $k \in [1, m]$ la fonction `polym`(p, k) renvoie le k-ième monôme de p. Si $k \notin [1, m]$ elle renvoie 0.

Exemple

On montre comment obtenir la décomposition du polynôme w en somme de monômes, et pour chaque monôme, comment extraire le coefficient, les littéraux et leurs exposants.

```
w=5*(a+x^2/3)^2
print "Décorticage de";w
print "  monôme      coef       littéraux et exposants"
for i=1,polymn(w)
  mono=polym(w,i)
  print mono;tab(12),norm(mono);tab(20);
  for j=1,polyln(mono)
    z=polyl(mono,j)
    print "   ";z;deg(mono,z);
  next j
  print
next i
```

Sortie (320 ms)

```
 Décorticage de  5*a^2 +10/3*a*x^2 +5/9*x^4
   monôme      coef       littéraux et exposants
   5*a^2       5          a  2
   10/3*a*x^2  10/3       a  1     x  2
   5/9*x^4     5/9        x  4
```

NUMF(p)

DENF(p)

NUM(p)

DEN(p)

V_fonctions Numérateur et Dénominateur

p

expr

Si p est une forme factorisée, nous écrivons :

$$p = P_1 \prod_{i=2}^{s} P_i^{k_i} = \frac{N}{D}$$

où $N = P_1 \prod_{k_i>0} P_i^{k_i}$ et $D = \prod_{k_i<0} P_i^{-k_i}$ sont des polynômes (sous forme factorisée) qui regroupent les facteurs de p suivant le signe de leurs exposants. Les fonctions `numf`(p) et `denf`(p) renvoient les formes factorisées N et D respectivement. Si p est sous forme polynôme, les fonctions `numf`(p) et `denf`(p) renvoient $N = p$ (sous forme factorisée) et $D = 1$ respectivement.

Les fonctions `num`(p) et `den`(p) renvoient respectivement N et D sous forme polynomiale.

Exemple

On considère une forme factorisée p et une forme polynomiale A.

```
p=173/23*(x-1)^-2*(x+y)*x^2*y^3/z
print numf(p)
print denf(p)
print num(p)
print den(p)
A=173/23*(x-1)*(y+3)
print numf(A)
print denf(A)
print num(A)
print den(A)
```

Sortie (405 ms)

```
173/23* [y]^3* [x]^2* [x +y]
 [z]* [x -1]^2
173/23*x^3*y^3 +173/23*x^2*y^4
x^2*z -2*x*z +z
173/23* [x*y +3*x -y -3]
1
173/23*x*y +519/23*x -173/23*y -519/23
1
```

FACTORN(p)

FACTORP(p [, k])

FACTORE(p [, k])

V_fonctions Facteurs

p

expr

k

entier*16 (par défaut $k = 1$)

Si le codage de p est une forme factorisée :

$$p = P_1 \prod_{k=2}^{s} P_k^{a_k}$$

le nombre de facteurs s est donné par `factorn`(p), le k-ième facteur P_k est donné par `factorp`(p, k) et son exposant a_k est donné par `factore`(p, k). Nous posons $a_1 = 1$, $P_k = 1$ et $a_k = 0$ pour $k \notin [1, s]$. Les fonctions `factorp` et `factore` sont ainsi étendues à toutes les valeurs de k.

Si p est codé sous forme polynomiale, la fonction `factorn`(p) renvoie 0, `factorp`(p, k) renvoie p si $k = 1$ et 1 sinon. La fonction `factore`(p, k) renvoie 1 si $k = 1$ et 0 sinon.

La fonction `factorn`(p) est également utilisable lorsque p est une expression flottante. Elle renvoie -1 ou -2 pour une expression réelle ou complexe respectivement.

Si W est une variable, `factorp`(W)=`peekws`(`ptr`(W)).

Exemple

Valeurs de `factorn` pour les divers cas possibles.

```
wexpr=(1+x)^3
print factorn(wexpr);peekws(ptr(wexpr));wexpr
wexpr=147*(x+y)^2/z
print factorn(wexpr);peekws(ptr(wexpr));wexpr
wexpr=2^.5
print factorn(wexpr);peekws(ptr(wexpr));wexpr
complex i
wexpr=(-27)^(1/3)
print factorn(wexpr);peekws(ptr(wexpr));wexpr
```

Sortie (430 ms)

```
  0  0  x^3 +3*x^2 +3*x +1
  3  3  147* [z]^-1* [x^2 +2*x*y +y^2]
 -1 -1  0.1414213562~ E+1
 -2 -2  0.1500000000~ E+1 +i*0.2598076211~ E+1
```

Exemple

On montre comment décomposer une forme factorisée.

```
factor
p=7/123*(3*x+1)^3*(17*x+2)^-2
```

```
print "Facteurs de";p
print "exposant  facteur"
for k=1,factorn(p)
  print factore(p,k);tab(8),factorp(p,k)
next k
```

Sortie (225 ms)

```
Facteurs de  7/123* [17*x +2]^-2* [3*x +1]^3
exposant  facteur
  1       7/123
 -2       17*x +2
  3       3*x +1
```

Exemple

Le programme suivant attend comme entrée un polynôme non constant. Il refuse les valeurs flottantes, les nombres exacts et les formes factorisées non réductibles à un polynôme.

```
do
  input "Entrer un polynôme >",p
  ift factorn(p)>0 p=formd(p)
  ift factorn(p)=0 ift polyln(p) exit
loop
print p
```

Exemple de dialogue

```
Entrer un polynôme >
17+pi
Entrer un polynôme >
124
Entrer un polynôme >
1/x
Entrer un polynôme >
(x^2+2*x+1)/(1+x)
  x +1
```

Calculs modulaires

Basic 1000d possède des fonctions effectuant des calculs modulaires. Les fonctions considérées ici utilisent le corps $\mathbf{Z}_p$ des entiers modulo p. Dans cette section p désigne un nombre premier. Les fonctions acceptent en général aussi des valeurs négatives de p, mais $p = 0$ et $p = \pm 1$ provoquent l'erreur Non Entier. Toutefois, certaines fonctions sont aussi valables pour p entier quelconque. On

dispose de nombreuses fonctions très performantes, agissant sur les nombres modulaires ainsi que sur les polynômes unip.

unip

Nous désignons ainsi des polynômes à un seul littéral, que nous notons par x, et à coefficients entiers. Ces polynômes sont considérés comme étant des éléments de $\mathbf{Z}_p[x]$, c'est à dire que les coefficients sont pris modulo p. En entrée les coefficients peuvent être des entiers quelconque, en sortie, sauf pour la fonction `mds`, ce sont des entiers dans $[0, p[$. Nous représentons les unip par A, B, Q, R, ... et v.

v

Nous réservons la lettre v pour désigner un unip intervenant lorsque les autres unip A, B, ... sont considérés modulo $v(x)$ et modulo p. Il sera souvent nécessaire que v soit premier avec A.

PRTST(m)

V_fonction Teste si m est premier

m

entier $m > 1$

La fonction `prtst` met 0 (faux) si m n'est pas premier et -1 (vrai) si m est premier. En réalité la valeur renvoyée n'est démontrée exacte que soit si c'est 0 (on est alors sûr que m est composé, mais les facteurs peuvent être très difficiles à déterminer) ou soit si $m < 10^8$. En effet `prtst` teste d'abord si m a des facteurs $< 10^4$ par division, et s'il n'en a pas, applique le test probabiliste de Miller & Rabin (voir N Koblitz (1987), chapitre 5). La probabilité que la réponse -1 soit fausse est inférieure à 10^{-40}. La fiabilité peut être augmentée en répétant l'appel, ainsi si deux appels fournissent -1, la probabilité d'erreur est plus faible que 10^{-80}.

Performances de PRTST

Voici les temps de calculs en secondes de `prtst`(m) lorsque m est le nombre qui s'écrit avec n chiffres 1 en décimal.

n	temps	
23	7	premier
53	1	composé
317	5876	premier
331	260	composé
1009	6843	composé
1511	22444	composé

Exemple

Le programme détermine les nombres premiers palindromes de 7 chiffres.

```
        forv i in (1,3,7,9)
          for j=0,9
            for h=0,9
              for k=0,9000,1000
                m=1000001*i+100010*j+10100*h+k
                if prtst(m)
                  print justr$(m,8);
                  n=n+1
                endif
          next k,h,j
        nextv
        print
        print "nb de solutions=";n
```

```
Sortie (368 s)
1003001 1008001 1022201 1028201 ...
...
nb de solutions=  668
```

PRIME(m)

V_fonction Nombre premier plus grand ou égal à m

m

entier $m > 1$

La fonction `prime` utilise `prtst`. Son résultat n'est donc que probablement vrai si $m > 10^8$, avec un taux d'erreur plus faible que 10^{-40}.

Exemple

La fonction suivante `premier`(k) donne le k-ième nombre premier. Le centième nombre premier est affiché.

```
        print premier(100)
        stop
premier:function(index j)
        value=1
        for j=1,j
          value=prime(value+1)
        next j
        return
```

Sortie (1115 ms)

```
541
```

Performances de PRIME

La boucle suivante, sur les nombres premiers inférieurs à 10^6 est effectuée en 20000 s environ.

```
        m=1
        while m<10^6
          m=prime(m+1)
```

```
wend
```

MDS(A, p)

V_fonction Reste modulo p

la fonction `mds` renvoie un unip égal à l'unip A modulo p, et à coefficients dans $[-p/2, p/2[$. Noter que la fonction `mod`(A, p) renvoie un unip équivalent à A et à coefficients dans $[0, p[$. Ces deux fonctions seront très utiles dans les calculs modulaires. Ainsi on écrira `mod(A*B,p)`, `mds(A+B,p)`, etc. pour effectuer les multiplications et additions modulaires.

Exemple

Les représentations signées modulo 29 des polynômes A, $2A$ et $(1+x)A$ sont affichées.

```
A=27*x^2+15*x+13
print mds(A,29)
print mds(2*A,29)
print mds(A*(1+x),29)
```

Sortie (125 ms)

```
-2*x^2 -14*x +13
-4*x^2 +x -3
-2*x^3 +13*x^2 -x +13
```

MDDIV(A, B, p)

MDMOD(A, B, p)

V_fonctions Division modulaire

Ces fonctions effectuent la division modulo p de l'unip A par l'unip B. Soient Q et R les unip tels que $A = B \times Q + R$ (modulo p), avec `deg`(R) < `deg`(B) et tels que les coefficients de Q et R soient dans $[0, p[$. Les fonctions `mddiv` et `mdmod` renvoient Q et R respectivement.

Exemple

La relation $A = B \times Q + R$ (mod 13) est vérifiée.

```
A=7*x^4+5*x+1
B=9*x^2+12
Q=mddiv(A,B,13)
R=mdmod(A,B,13)
print Q
print R
print mod(A-B*Q-R,13)
```

Sortie (110 ms)

```
8*x^2 +11
5*x +12
0
```

MDGCD(A, B, p)

V_fonctions Pgcd

La fonction `mdgcd` calcule le pgcd modulaire normalisé des unips A et B. Elle renvoie l'unip W de degré maximum qui divise à la fois A et B, et tel que `polyn`(W)=1. Autrement dit, si le degré de W est n, W est de la forme $W = x^n + c_{n-1}x^{n-1} + \cdots + c_0$. Un des unip entré peut être nul, ainsi `mdgcd`(A, 0, p) renvoie l'unip normalisé proportionnel à A. Cela correspond à la convention que tout polynôme divise 0.

Exemple

On vérifie que le pgcd modulaire P de A et B divise A et B dans $\mathbf{Z}_7[x]$. La dernière instruction normalise $3P$.

```
A=3*x^4+x^3+4*x^2+4*x+3
B=3*x^4+3*x^3+x+3
P=mdgcd(A,B,7)
print P
print mdmod(A,P,7);mdmod(B,P,7)
print mdgcd(3*P,0,7)
```

Sortie (165 ms)

```
x^2 +5*x +2
0  0
x^2 +5*x +2
```

MDPWR(A, n, v, p)

V_fonction Puissance doublement modulaire

n

entier ($n < 0$ accepté si `mdgcd`(A, v, p) = 1)

MDINV(A, v, p)

V_fonction Inverse doublement modulaire

La fonction `mdpwr` calcule la puissance n-ième de l'unip A modulo l'unip v et modulo p. Elle renvoie l'unip W à coefficients dans $[0, p[$ et de degré `deg`(W) < `deg`(v) tel que $W = A^n$ modulo v et modulo p. L'entier n n'est pas limité à $[-2^{15}, 2^{15}[$ comme dans l'expression A^n, mais peut être aussi grand que 10^{19723}. La fonction `mdinv`(A, v, p) qui est identique à `mdpwr`(A, -1, v, p) calcule A^{-1} (mod v, mod p).

Exemple

Le programme calcule $W = A^{500000}$ et $U = A^{-1000000}$ (mod $x^7 + 1$, mod 17). La fonction `mdmod` permet de vérifier que $W^2 - U^{-1} = 0$ (mod $x^7 + 1$, mod 17).

```
A=2*x^3+1
v=x^7+1
W=mdpwr(A,10^6/2,v,17)
U=mdpwr(A,-10^6,v,17)
```

```
        print W
        print U
        print mdmod(W^2-mdinv(U,v,17),v,17)
```

Sortie (2380 ms)

```
8*x^6 +15*x^5 +13*x^4 +16*x^3 +15*x +9
8*x^6 +6*x^5 +4*x^4 +12*x^3 +9*x^2 +14*x +12
0
```

MDPWRE(m, n, p)

V_fonction Puissance modulaire

m, n

entiers

La fonction `mdpwre` calcule la puissance n-ième de l'entier m modulo p. Elle renvoie un nombre dans $[0, p[$ égal à m^n modulo p, où n est un entier qui peut être négatif et qui peut atteindre 10^{19723}. Si $n \geq 0$, p non premier est accepté.

Exemple

Le programme montre que $a = 3^{10^{100}} = 16 \pmod{19}$ et $b = 3^{-3\times 10^{100}} = 7 \pmod{19}$. On vérifie la relation $a^3 b = 1 \pmod{19}$ à l'aide de la fonction `modr`.

```
        m=3
        p=19
        a=mdpwre(m,10^100,p)
        b=mdpwre(m,-3*10^100,p)
        print a;b;modr(a^3*b,p)
```

Sortie (690 ms)

```
16  7  1
```

PRINV(m, p)

V_fonction Inverse modulaire

m, p

entiers

La fonction `prinv` calcule l'inverse de l'entier m modulo p, si m et p sont premiers entre-eux. Elle renvoie le nombre n dans $[1, p[$ tel que $mn = 1$ (modulo p). La forme `prinv`(m, p) est alors équivalente à `mdpwre`(m, -1, p). Si par contre pgcd(m, p) $\neq 1$, l'inverse n'existe pas, `prinv`(m, p) renvoie 0, mais `mdpwre`(m, -1, p) sort en erreur.

Exemple

Le programme demande deux entiers et écrit avec eux l'identité de Bezout.

```
        print /c/"Identité de Bezout"
        print "Entrer deux entiers"
        input a,b
        a=abs(a)
        b=abs(b)
```

```
        ift a>b exg a,b
        p=gcdr(a,b)
        if a=p
          print a;"=";a
        else
          ap=prinv(a/p,b/p)
          bp=(p-a*ap)/b
          print ap;"  *";a;"    ";bp;"  *";b;"=";p
        endif
```

Exemple de dialogue :

```
Entrer deux entiers
INPUT >
41
INPUT >
127
  31  *  41     -10  *  127=  1
```

MDFF(A, p)

V_fonction Factorisation

La fonction `mdff` factorise complètement l'unip A dans $\mathbf{Z}_p[x]$. Elle renvoie une forme produit d'unip irréductibles (modulo p), égale à A modulo p. Pour $p = 2$ (et seulement pour $p = 2$), il y a des polynômes que la fonction ne sait pas complètement factoriser. Dans ce cas, il n'y a pas de retour de la fonction (sauf par Break).

Exemple

Le polynôme $x^9 + 1$ est factorisé modulo 19.

```
print mdff(x^9+1,19)
```

Sortie (755 ms)

```
    [x +17]* [x +16]* [x +11]* [x +9]* [x +7]* [x +6]* [x +5]* [x +4]*
  [x +1]
```

MDSMP(U, p)

V_fonction Réduction en éléments simples modulo p

U

forme factorisée décrivant la fraction à réduire

Soient $n-1$ unip A_2, A_3, ..., A_n, premiers deux à deux modulo p. La réduction en éléments simples (mod p) de

$$\frac{1}{A_2 \times A_3 \times \cdots \times A_n}$$

consiste à déterminer les $n-1$ unip B_2, B_3, ..., B_n tels que $\mathtt{deg}(B_i) < \mathtt{deg}(A_i)$ pour $i = 2, 3, \ldots, n$ qui vérifient :

$$\frac{1}{A_2 \times A_3 \times \cdots \times A_n} = \frac{B_2}{A_2} + \frac{B_3}{A_3} + \cdots + \frac{B_n}{A_n} \pmod{p}.$$

La fonction `mdsmp` effectue cette réduction. En entrée, U doit être la forme factorisée :

$$U = A_1 \times A_2 \times A_3 \times \cdots \times A_n$$

où le facteur constant A_1 = `factorp`(U, 1) et les exposants `factore`(U, i) ne jouent aucun rôle. Les $n-1$ unip B_2, B_3, ..., B_n sont renvoyés au moyen d'une forme non standard W = `mdsmp`(U, p) codée comme une forme produit. L'unip B_i s'obtient par B_i = `factorp`(W, i). La forme W est non standard car les unip B_i ne sont ni premiers deux à deux, ni ordonnés de façon habituelle. Il est cependant possible de l'afficher.

Exemple

Réduction de $1/(A_2 \times A_3)$

On forme U en mode `factor`, on en tire A_2 et A_3 et on vérifie qu'ils sont premiers modulo 29. La réduction :

$$\frac{1}{A_2 \times A_3} = \frac{B_2}{A_2} + \frac{B_3}{A_3} \qquad \text{(mod 29)}.$$

équivaut à l'identité de Bezout $A_2B_3 + A_3B_2 = 1 \pmod{29}$ que l'on vérifie à l'aide de `mod`.

```
factor
U=(x^2+1)*(x+2)
A2=factorp(U,2)
A3=factorp(U,3)
p=29
print mdgcd(A2,A3,29)
print A2; A3
W=mdsmp(U,p)
B2=factorp(W,2)
B3=factorp(W,3)
print B2;B3
print mod(A2*B3+A3*B2,p)
```

Sortie (215 ms)

```
1
 [x^2 +1]   [x +2]
 [23*x +12]  6
1
```

Fonctions aléatoires

RANDOM(p)

RANDOM(p, x, k)

V_fonction Entier ou polynôme aléatoire

p

entier ($|p| > 1$)

x

littéral

k

entier

La forme `random`(p) revoie un entier aléatoire dans $[0, |p|[$. La forme `random`(p, x, k) renvoie un polynôme aléatoire unilittéral en x de degré k à coefficients dans $[0, |p|[$.

Exemple

```
print random(2^150)
print random(2^10,x,5)
```

Sortie (145 ms)

```
1128039388263672206571894779001217997184326302
665*x^5 +754*x^4 +673*x^3 +633*x^2 +714*x +172
```

RND

RND(virchaîne)

V_fonction Flottant aléatoire dans $[0, 1]$

La fonction `rnd` calcule `float`(`random`(2^k)/2^k) où k est le nombre de bits utilisés dans la représentation des nombres flottants. Dans la deuxième forme, virchaîne est calculée mais non utilisée.

Exemple

```
precision 50
print rnd
```

Sortie (240 ms)

```
0.31384769492959736997016869069091802508684499289800~
```

RANDOMIZE

RANDOMIZE 0

RANDOMIZE r

Commande Initialise le générateur de nombres aléatoires

r

réel

Le générateur de nombres aléatoires du Basic 1000d utilise une méthode due à G J Mitchell et D P Moore (1958) basée sur la suite $S_n = S_{n-24} + S_{n-55}$ (modulo 2^{16}). Après initialisation des 55 nombres S_1, S_2, …, S_{55} par des

nombres non tous pairs, la suite S_n, qui a une période plus grande que $2^{55} - 1 = 36028797018963967$, peut être considérée comme aléatoire. La commande `randomize` r, avec $r \neq 0$, initialise S_1, S_2, ..., S_{55} de manière reproductible. On obtiendra alors, dans un programme donné, les mêmes valeurs dans `rnd` et `random`. Noter cependant que ces fonctions sont utilisées implicitement par certaines fonctions du Basic (comme par exemple `formf`)

Exemple

Ce programme produit les mêmes résultats chaque fois qu'il est exécuté. Cependant les 2 valeurs de `random`(1000) diffèrent parce que le générateur de nombre aléatoire est appelé par `formf`.

```
randomize 1
w=formf(1-x^2)
print random(1000)
randomize 1
print random(1000)
print rnd
```

Sortie (235 ms)

```
502
184
0.6058745258~
```

Les commandes `randomize` (sans argument) et `randomize` r (avec $r = 0$) initialisent la suite S_n avec le compteur 200 Hz en $4BA. Les programmes ne sont alors plus reproductibles.

Exemple

Tests du χ^2

Ce programme montre que les suites aléatoires obtenues par `random` se comportent très bien dans les tests du χ^2.

```
print "Tests KHI^2: O=mauvais (0-1%)  o=douteux (1-5%)
  .=bon"
print "Par ligne, moins d'un O et cinq o correspondent
 à un test très réussi"
var T(9)
FORMAT 3
for R=1,15
  RANDOMIZE R
  print "R=";justl$(R,2);
  N=50'il vaut mieux augmenter N
  FOR J=1,50
    FOR I=0,9
      T(I)=0
    next I
    FOR I=1,N
      VADD T(RANDOM(10)),1
```

```
        next I
        KHI=FLOAT(SUM(I=0,9 OF T(I)^2)*10/N-N)
        SELECT CASE KHI
        CASE IN [3.325,16.92]
          print ".";
        CASE IN [2.088,21.67]
          print "o";
        CASE OTHERS
          print "O";
        endselect
      next J
      print
    next R
```

Exercices de programmation

Exercice Fibo

Nous avons donné dans le chapitre d'introduction un programme calculant exactement le millième nombre de la suite de Fibonacci 1, 1, $2 = 1 + 1$, $3 = 2 + 1$, $5 = 3 + 2$, ... (Le résultat est un nombre de 209 chiffres). Ecrire un programme calculant ce nombre sans utiliser une fonction à appel récursif (pour minimiser le temps de calcul).

Exercice Grandissime

Les nombres de Mersenne M_p sont les nombres premiers de la forme $2^p - 1$. Un entier N est appelé nombre parfait s'il est égal à la somme de ses diviseurs comme par exemple $6 = 1 + 2 + 3$. Un théorème célèbre (Euclide et Euler) montre que les nombres parfaits pairs sont en correspondance bijective avec les nombres de Mersenne. Les nombres parfaits pairs sont l'ensemble des nombres de la forme $2^{p-1}M_p$ où $M_p = 2^p - 1$ est un nombre de Mersenne.

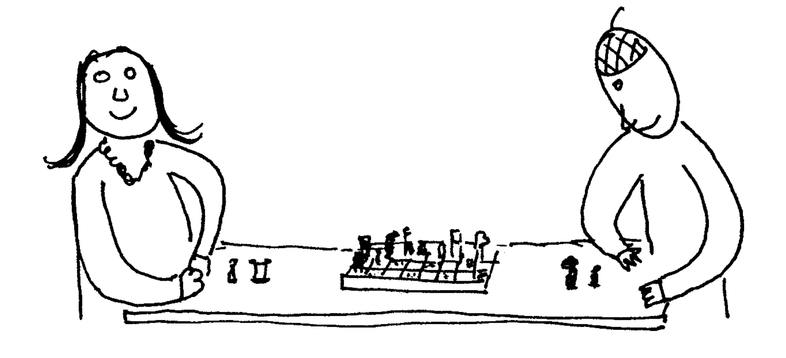
En 1979, le vingt-septième nombre de Mersenne connu, $2^{44497} - 1$, était aussi le plus grand nombre premier connu (D. Slowinski). Mais la liste des nombres de Mersenne connus s'allonge presque tous les ans $2^{86243} - 1, \ldots, 2^{216091} - 1$ (en 1985), ... ?

Ecrire les nombres parfaits correspondants. Comme ces nombres sont plus grands que 2^{65520}, il faut effectuer des calculs en multiprécision. En Basic 1000d il existe une méthode très simple, dans laquelle les grands nombres sont codés par des polynômes.

14

Calculs conditionnels

Nombres complexes



Basic 1000d permet de calculer modulo un nombre réel ou modulo un ensemble de polynômes. Ce type de calcul s'obtient en déclarant des conditions à l'aide de la commande `cond`. La commande `complex i` permettant de calculer en nombres complexes est apparentée aux calculs conditionnels. Elle impose en effet la condition $i^2 = -1$ au littéral i.

COND p [, x]

COND e

Commande Spécification des conditions

p

poly

x

littéral [par défaut $x = \texttt{polyl}(p)$]

e

nombre exact

On peut imposer `s_cond` (variable d'état modifiable) conditions. Le Basic 1000d réduit les expressions suivant les conditions en vigueur seulement aux moments suivants :

• Avant toute assignation d'une expression. Noter que les commandes `read` et `input`, qui effectuent aussi des assignations implicites sont concernées par les conditions.

• Dans une exprc des formes suivantes :

```
expra = expra
expra <> expra
expra IN v_ensemble
expra NOT IN v_ensemble
```

les conditions sont utilisées pour déterminer l'égalité ou la non-égalité. Cela concerne aussi les comparaisons (en deux morceaux) effectuées dans la structure `select case`.

• Dans les commandes :

```
ift x ...
if x
  while x
  until x
```

l'expression x est réduite suivant les conditions.

Exemple

La commande `complex` impose la condition $i^2 + 1 = 0$.

```
complex i
if i^2+1
  anormal
else
  print "ok"
```

```
          endif
```

Sortie (25 ms)

```
ok
```

On notera que, par contre, les conditions ne sont pas utilisées dans les cas suivants :

• Dans les exprc qui font intervenir des inégalités.

• Dans `print`, qui n'effectue pas d'assignation et calcule et écrit les expressions sans tenir compte des conditions.

Exemple

Le programme ci-dessous fonctionne correctement. Par contre, on ne peut pas remplacer dans ce programme la troisième ligne par `case <3` qui produit une erreur Comparaison (les membres de la comparaison doivent être réels). Cette même erreur serait également produite par l'instruction `if s^2<3`.

```
cond s^2-2
select s^2
case =2
  print "ok"
endselect
```

Sortie (30 ms)

```
ok
```

Examinons maintenant comment s'effectue la prise en compte des conditions sur une expression q. Les conditions écrites avec un nombre e impliquent le calcul modulo e de tous les nombres (dans ce cas les calculs sous forme factorisée sont interdits). La valeur q est alors remplacée par `mod`(q, e). Chaque commande `cond` p, x implique la simplification $q =$ `mod`(q, p, x). Ce calcul est plus rapide si p est un polynôme de degré n de la forme $p = x^n + c_{n-1}x^{n-1} + \cdots$ (c'est à dire tel que `coef`(p, x, n)=1). Les conditions restent valables jusqu'à effacement par `clear cond` ou `clear`.

Exemple

Calcul modulo 2

```
cond 2
W=5*X+7*Y-Z+2*A'il faut effectuer une assignation
print W
```

Sortie (95 ms)

```
 X +Y +Z
```

Les calculs modulo un nombre premier p peuvent ainsi être effectués en imposant la condition `cond` p. Cependant dans ce cas il vaut mieux utiliser les fonctions modulaires du Basic, qui sont beaucoup plus performantes, et ne pas imposer de condition. Au lieu du programme précédent, il vaut mieux utiliser (en particulier si on a aussi besoin de nombres non modulaires) :

```
print mod(5*X+7*Y-Z+2*A,2)
```

Sortie (30 ms)

```
 X +Y +Z
```

Exemple

Les conditions `cond` permettent de travailler sur les irrationnels. Ainsi pour traiter exactement des expressions contenant $\sqrt{2}$, on introduit le littéral `sqrt2` et la condition `sqrt2`^2=2.

```
cond sqrt2^2-2
W2=sqrt2^2
W3=sqrt2^3
print W2; W3
```

Sortie (40 ms)

```
2  2*sqrt2
```

Exemple

Les conditions sont appliquées une seule fois, et dans l'ordre des commandes `cond`. Il peut donc être nécessaire d'effectuer une ou plusieurs assignations comme `y=y` pour simplifier complètement une expression. C'est en prévision de cas semblables à la deuxième partie de l'exemple que les conditions sont appliquées une seule fois (comme dans `subsr`) et non de façon cyclique (comme dans `subsrr`). En effet, le programme entrerait alors dans une boucle sans fin normale.

```
cond v^2+v+1,v
cond u^2-v*u+3,u
y=u^2*v
print y
y=y
print y
cond a-b^2,a
cond b-a^2,b
y=a
print y
y=y
print y
```

Sortie (280 ms)

```
 v^2*u -3*v
-v*u -3*v -u
 a^4
 a^16
```

CLEAR COND

Commande Efface les conditions

La commande `clear cond` efface toutes les conditions définies par `cond`.

Exemple

La commande `complex` n'est pas concernée par `clear cond`.

```
complex i
cond t^2+1
```

```
clear cond
w=i^2+t^2
print w
```

Sortie (45 ms)

```
t^2 -1
```

COMPLEX i

COMPLEX r

Variable d'état Mode complexe ou réel

i

littéral

r

expression égale à zéro

La forme `complex i` permet les calculs en nombre complexes. Le littéral `i` qui représente $\sqrt{-1}$, est désigné sous le nom de littéral complexe. La deuxième forme (ou `clear`) permet de revenir en mode réel. En tant que variable d'état, `complex` renvoie 0 en mode réel et le littéral complexe en mode complexe. Les expressions complexes exactes sont traitées presque toujours comme des expressions algébriques réelles, le littéral complexe étant sur le même plan que les autres littéraux. Toutefois les particularités des nombres complexes sont prises en compte dans les cas suivants :

• Lors du calcul des conditions, en particulier lors d'une assignation, la réduction $i^2 \to -1$ est effectuée. De plus dans les expressions, même factorisées, les facteurs numériques complexes sont réduits à un seul facteur complexe d'exposant 1.

Exemple

L'expression complexe est réduite à sa forme canonique seulement après une assignation.

```
complex i
print i^2
d=i^2
print d
```

Sortie (40 ms)

```
 i^2
-1
```

Exemple

Réduction à un seul facteur nombre complexe.

```
complex i
factor
w=(3+i)^7*(2*i-3)^-2/(a+i*b)
print w
```

Sortie (215 ms)

```
-8/169* [1453*i +4929]* [i*b +a]^-1
```

• La V_fonction `formf`(p) peut factoriser des polynômes avec nombres complexes.

Exemple

```
complex j
W=(4*x-3*j)*(4*x+3*j)
print W
print formf(W)
print formf(x^4+1)
```

Sortie (680 ms)

```
 16*x^2 +9
- [3*j -4*x]* [3*j +4*x]
- [j -x^2]* [j +x^2]
```

Cependant à la différence des calculs rationnels réels, `formf` ne sait pas factoriser toutes les expressions factorisables en complexes.

• La fonction `dive` et la commande `vdive` fonctionnent pour les polynômes en complexes.

Exemple

```
complex i
print dive(a^2+b^2,a+i*b)
print dive(1,3*i+1)
```

Sortie (75 ms)

```
-i*b +a
-3/10*i +1/10
```

• Ailleurs le traitement des expressions complexes peut être incomplet. En particulier, la règle qu'une expression factorisée ne comporte que des facteurs deux à deux premiers entre eux, n'est valable qu'en réels. Des expressions factorisées comme :

```
(a+i*b)/(a^2+b^2)
```

où les deux facteurs ne sont pas premiers sur **C**, ne sont pas automatiquement simplifiées.

• La fonction `root` ne fonctionne pas en complexes.

Exemple

Le Basic ne reconnaît pas que $\sqrt{2i} = \pm(1 + i)$.

```
complex i
print root(2*i,2)
```

Sortie (40 ms)

```
 0
```

Si plusieurs commandes `complex i` sont exécutées, seule la dernière est valable, les littéraux complexes antérieurs redeviennent des littéraux ordinaires. Après `complex` 0 (ou `complex phantom` qui lui est équivalent), l'ancien littéral complexe redevient un littéral ordinaire.

Les fonctions complexes décrites ci-dessous exigent que le littéral complexe soit défini.

FORMC(w)

V_fonction Forme complexe standard

CC(w)

V_fonction Conjugué complexe

RE(w)

IM(w)

V_fonction Parties réelle et imaginaire

CXNORM(w)

V_fonction Norme

w

expr

Lorsque w est une expr exacte, ces fonctions renvoient aussi une forme exacte. La fonction `formc` regroupe tous les facteurs contenant le littéral complexe (i pour fixer les idées) et renvoie une expression, égale à w, de la forme $(A+iB)C$ où A et B sont des polynômes, C une expression factorisée, et où ni A, ni B, ni C ne contiennent i. Si `formc` renvoie la forme ci-dessus, les fonctions `cc`, `re`, `im` et `cxnorm` renvoient respectivement le nombre complexe conjugué $(A-iB)C$, la partie réelle AC, la partie imaginaire BC et la norme $(A^2+B^2)C^2$. Dans ces calculs A, B et C sont donc interprétés comme étant réels.

Lorsque w est un flottant, les fonctions `formc` et `cc` renvoient un flottant complexe (de type -2, même si `im`(w)=0), et les fonctions `re`, `im` et `cxnorm` renvoient un flottant réel (de type -1).

Exemple

Cas exact. L'interprétation de `re`(w) comme partie réelle de w tombe à l'eau si on effectue ultérieurement la substitution `a=i`, par exemple.

```
complex i
w=1/(a+i*b)
print formc(w)
print re(w)
print im(w)
print cc(w)
print cxnorm(w)
```

Sortie (510 ms)

```
 - [a^2 +b^2]^-1* [i*b -a]
   [a]* [a^2 +b^2]^-1
 - [b]* [a^2 +b^2]^-1
   [a^2 +b^2]^-1* [i*b +a]
   [a^2 +b^2]^-1
```

Exemple

Cas flottant

```
complex i
w=1/3+i/4~
print w
print re(w);im(w)
print cc(w)
print cxnorm(w)
```

Sortie (215 ms)

```
0.3333333333~ +i*0.2500000000~
0.3333333333~  0.2500000000~
0.3333333333~ -i*0.2500000000~
0.1736111111~
```

CXABS(w)

V_fonction Valeur absolue exacte en complexe

w

expr

Considérons d'abord le cas où w est une expr exacte et posons $y =$ `cxnorm`(w). Si y est le carré d'une expression rationnelle, la fonction `cxabs` renvoie `root`$(y,$ $2)$, qui correspond au calcul exact du module $|w|$. Sinon, il y a une sortie erreur Non Rationnel.

Lorsque w est un nombre flottant, `cxabs`(w) est identique à `cabs`(w) et correspond au calcul en flottant de $|w|$.

Exemple

```
complex i
print cxabs(3+4*i)
print cxabs(2*t*i+1-t^2)
print cxabs(1~+i)
```

Sortie (220 ms)

```
5
 [t^2 +1]
0.1414213562~ E+1
```

CABS(w)

CARG(w)

V_fonctions Module et argument

w

nombre complexe

Les fonctions `cabs` et `carg` renvoient un nombre réel flottant, même lorsque le résultat pourrait être calculé exactement. Les valeurs renvoyées correspondent au module ρ et à l'argument θ du nombre complexe $w = \rho e^{i\theta}$. La fonction `carg` renvoie la détermination $\theta \in]-\pi, \pi]$ de l'argument. Les relations suivantes sont vérifiées :

```
cabs(w)=sqr(Re(w)^2+Im(w)^2)
```

```
carg(w)=atn2(Im(w),Re(w))=Im(log(w))
```

Exemple

```
complex i
print cabs(3+4*i);carg(1+i)
```

Sortie (115 ms)

```
0.5000000000~ E+1  0.7853981634~
```

CXINT(w)

V_fonction Plus proche entier de Gauss

w

nombre complexe

Si $w = a + ib$ désigne la décomposition en parties réelle et imaginaire du nombre complexe w, la fonction `cxint`(w) renvoie le nombre complexe exact `cint`(a)+i*`cint`(

Exemple

```
complex i
print cxint(3.7-2.4*i)
```

Sortie (70 ms)

```
-2*i +4
```

CXDIV(x, y)

CXMOD(x, y)

V_fonctions Division entière

x, y

nombres complexes

Les fonctions `cxdiv` et `cxmod` effectuent la division entière de x par y en complexe. Elles renvoient respectivement les nombres q et r tels que $x = q \times y + r$ et q = `cxint`(x/y). Des arguments x et y, autres que des nombres complexes sont acceptés si x/y est un nombre. La valeur q = `cxdiv`(x, y) est un nombre complexe exact. La valeur r = `cxmod`(x, y) est une expression exacte si x et y sont tous deux exacts, et un nombre flottant sinon.

Exemple

Effectue la division de x par y et vérifie que $x = qy + r$.

```
complex i
x=2^20+i*3^15
y=5^7+i*7^3
q=cxdiv(x,y)
r=cxmod(x,y)
v=x-q*y-r
print xa̲;ya̲;qa̲;ra̲;v
```

Sortie (695 ms)

```
14348907*i +1048576
343*i +78125
```

```
  184*i +14
-30895*i +17938
  0
```

CXGCD(a {, b})

CXINV(a, b)

V_fonctions Pgcd d'entiers de Gauss

a, b, ...

entiers de Gauss

Un entier de Gauss est un nombre complexe dont les parties réelle et imaginaire sont des entiers. Il doit être donné sous forme exacte et non flottante. Si a et b sont deux entiers de Gauss, on dit que a divise b s'il existe un entier de Gauss c tel que $b = ac$. La divisibilité peut être testée par `cxmod`(b, a), qui est nul si et seulement si a divise b. Le pgcd de a et b, $g =$ `cxgcd`(a, b), est un entier de Gauss de norme `cxnorm`(g) maximum qui divise a et b. Le pgcd est unique, à une unité ± 1 ou $\pm i$ près. Il existe des entiers de Gauss x et y, tels que $ax + by = g$ (Identité de Bezout). La fonction `cxinv` permet d'obtenir des entiers de Gauss vérifiant cette égalité, par :

```
x=cxinv(a,b)
y=(cxgcd(a,b)-a*x)/b
```

Exemple

La méthode suivante permet d'écrire certains nombres premiers comme somme de deux carrés. Pour cela le nombre premier p doit diviser un nombre de la forme $w^6 + 1$ (par exemple $p = 4562284561$ qui divise $1024^6 + 1$). Pour écrire $p = u^2 + v^2$, il suffit de trouver un entier de Gauss, autre qu'une unité, qui soit un facteur de p, puisque on en déduit alors une factorisation $p = (u + iv)(u - iv) = u^2 + v^2$. Comme :

$$w^6 + 1 = (w^2 + 1)((w^2 - 1)^2 + w^2)$$

p divise un des deux facteurs à droite, le deuxième pour l'exemple choisi. Ce deuxième facteur s'écrivant $(w^2 - 1 + iw)(w^2 - 1 - iw)$, on obtient $g = u + iv$ par le programme suivant :

```
complex i
p=4562284561
w=1024
g=cxgcd(p,w^2-1+i*w)
print using "#_^2 _+ #_^2 = #",abs(re(g)),abs(im(g)),p
```

Sortie (480 ms)

```
65281^2 + 17340^2 = 4562284561
```

15

Calculs approchés



Les fonctions décrites dans ce chapitre renvoient un nombre réel ou complexe flottant. L'erreur relative sur le résultat correspond à la précision demandée par la commande `precision` ou `precision2`.

En flottant aussi

Nous rappelons ici, sans détails mais en donnant des exemples, des fonctions qui ont été étudiées précédemment comme fonctions exactes, et qui peuvent aussi s'utiliser avec des nombres flottants.

ABS(p)

Valeur absolue du réel p

Exemple

```
print abs(-17/2~)
```

Sortie (35 ms)

```
0.8500000000~ E+1
```

MIN(p, q {, r})

Minimum des réels p, q, ...

Exemple

Il suffit que l'un des arguments soit flottant pour que le résultat soit flottant, même si la valeur minimum a été donnée en exact.

```
print min(2000~,1000)
```

Sortie (45 ms)

```
0.1000000000~ E+4
```

MAX(p, q {, r})

Maximum des réels p, q, ...

Exemple

Détermine le maximum de 100 nombres tirés au hasard.

```
randomize 7
m=0
for i=1,100
  m=max(m,rnd)
next i
print m
```

Sortie (8625 ms)

```
0.9985580577~
```

MODR(p, q)

MODS(p, q)

DIVR(p, q)

Division entière de deux réels

Exemple

Lorsqu'un des arguments au moins est flottant, le reste de la division (positif `modr`, ou signé `mods`) est un flottant. Par contre, le quotient `divr` est toujours un entier.

```
w=10
print divr(12,pi);modr(12,pi);mods(12,pi)
```

Sortie (160 ms)

```
3  0.2575222039~ E+1 -0.5663706144~
```

PPWR(p, k)

Symbole de Pochhammer $p^{(k)}$

Le premier argument p peut être flottant (réel ou complexe).

Exemple

Calcule une valeur approchée de factorielle 1000.

```
print ppwr(1000~,1000)
```

Sortie (1920 ms)

```
0.4023872601~ E+2568
```

Noter que ce calcul est plus rapide que le calcul et la transformation en flottant de l'entier ppwr(1000), qui s'écrit avec 2568 chiffres.

```
print float(ppwr(1000))
```

Sortie (5930 ms)

```
0.4023872601~ E+2568
```

V_fonctions approchées

Les V_fonctions approchées acceptent que les arguments soient donnés sous forme exacte, mais le temps de calcul est alors plus long car il y a d'abord conversion en flottant.

FLOAT(p)

FLOAT(p, q)

V_fonction Conversion en flottant

p, q

nombres réels ou complexes

La conversion d'exact en flottant est effectuée à la précision courante. La forme `float`(p, q) renvoie le nombre complexe flottant $p + iq$, i désignant le littéral complexe. Elle permet la conversion en complexe puisque même lorsque sa partie imaginaire est nulle, la valeur renvoyée est un nombre complexe (de type -2). La forme `float`(p) renvoie un nombre flottant, réel si possible, approchant p.

Exemple

```
complex i
w=float(2,i)
print w
print float(w)
```

Sortie (80 ms)

```
0.1000000000~ E+1 +i*0~
0.1000000000~ E+1
```

FSUBS(p $\{$, xi = fi $\}$)

V_fonction Substitution flottante

p

expr exacte

xi

littéral

fi

nombre réel ou complexe

La fonction `fsubs`(p, $x_1 = f_1$, $x_2 = f_2$, $\ldots$, $x_n = f_n$) renvoie la valeur flottante de l'expression p dans laquelle on a substitué le littéral x_1 par f_1, le littéral x_2 par f_2, ... Tous les n littéraux de p, sauf le littéral complexe, doivent être spécifiés dans l'appel de la fonction.

Exemple

La substitution flottante de la racine $x = 1/7$ ne donne pas zéro par suite des approximations, à la différence de la substitution exacte effectuée par `subs`. Si plusieurs substitutions dans une même expression w doivent être effectuées, les calculs seront plus rapides si w est fournie à `fsubs` sous forme produit de polynômes comme ici. En effet, la forme produit est plus rapide pour la conversion flottante parce que tous les nombres sont entiers, mis à part un facteur rationnel.

```
factor
w=91*x^2 -20*x +1
print fsubs(w,x=1/7)
```

```
print subs(w,x=1/7)
```

Sortie (100 ms)

```
-0.1776356839~ E-14
 0
```

SQR(p)

V_fonction Racine carrée

p

nombre complexe

La fonction sqr(p) renvoie $\sqrt{p}$.

Exemple

Calcul de $\sqrt[4]{-4}$

La valeur de la racine est obtenue par $\sqrt{\sqrt{-4}}$.

```
complex i
print sqr(sqr(-4))
```

Sortie (325 ms)

```
1.0000000000~ +i*1.0000000000~
```

Pour calculer la racine quatrième en complexes, il est toujours plus rapide d'utiliser l'exponentielle. En effet la forme `sqr(p)`, en complexe, est calculée par `p^(1/2)`.

```
complex i
print (-4)^(1/4)
```

Sortie (180 ms)

```
1.0000000000~ +i*1.0000000000~
```

Par contre, en réel, l'extraction de deux racines carrées, qui est effectuée par une procédure spéciale, est plus rapide que l'exponentielle.

```
print sqr(sqr(5))
```

Sortie (40 ms)

```
0.1495348781~ E+1
```

```
print 5^(1/4)
```

Sortie (90 ms)

```
0.1495348781~ E+1
```

PI

V_fonction Valeur de $\pi \approx 3.14$

Exemple

Ecrit `pi` avec 50 chiffres.

```
precision 50
print pi
```

Sortie (80 ms)

```
0.31415926535897932384626433832795028841971693993751~ E+1
```

Exercice piR2

Calculer la surface d'un cercle de rayon 1/7 avec 100 chiffres exacts.

EXP(p)

V_fonction Exponentielle

p

nombre réel ou complexe

La fonction `exp` renvoie e^p. L'argument est limité par $\mathrm{Re}(p) < 22500$ (environ).

Exemple

```
print exp(22500)
complex i
print exp(22500+i)
```

Sortie (340 ms)

```
0.4225156727~ E+9772
0.2282861922~ E+9772 +i*0.3555346792~ E+9772
```

LOG(p)

LOG(p, a)

LOG10(p)

V_fonctions Logarithme

p, a

Nombres réels ou complexes

La forme `log`(p) renvoie le logarithme naturel $\log p$. La forme `log`(p, a) renvoie $x = \log_a p = \log p / \log a$, le logarithme en base a de p. Elle donne une solution x de l'équation $p = a^x$. La fonction `log10`(p), qui équivaut à `log`(p, 10), renvoie $\log_{10} p$, le logarithme en base 10 de p.

En réel, p et a doivent être dans $[10^{-9000}, 10^{9000}]$. En complexe, c'est `cxnorm`(p) qui doit être dans cet intervalle. La fonction `log`(p) renvoie alors la détermination x du logarithme telle que $\mathrm{Im}(x) \in]-\pi, \pi]$.

Au voisinage de 1, `log`(p) devient très imprécis en valeur relative, même lorsque p est donné en exact. Cette erreur provient de la conversion de p en flottant. Pour $p \ll 1$ ou $p \gg 1$, la précision relative du résultat correspond à la précision en cours. Pour p réel quelconque, l'erreur absolue Δx sur $x =$ `log`(p) vérifie la relation :

$$\frac{\Delta x}{1 + |x|} < 2^{-\mathrm{precision2}}.$$

.

Exemple

Le programme vérifie l'inégalité ci-dessus, pour des nombres $p = \exp w$ aléatoires, dont la moitié sont voisins de 1.

```
for i=-19000,22000,10
  forv w in (i+rnd,1/(i+rnd))
```

```
            e=abs(log(exp(w))-w)
            y=max(e/(1+abs(w)),y)
          nextv
        next i
        print y;2~^-precision2
```

Sortie (1342 s)

```
0.1547846325~ E-13  0.1136868377~ E-12
```

EXP1(r)

LOG1(r)

V_fonctions Logarithme et exponentielle

r

réel

Les fonctions `exp1` et `log1`, inverses l'une de l'autre, sont définies par `log1`$(r) = \log(1+r)$ et `exp1`$(r) = \exp(r)-1$. La fonction `log1`(r), au contraire de `log`$(1+r)$, a une erreur relative correspondant à la précision en cours, même pour r petit devant 1. Le calcul de `exp`$(r) - 1$ est beaucoup plus précis en utilisant `exp1`(r) lorsque r est très petit devant 1.

Exemple

Le calcul de $(1+\frac{1}{x})^x$ pour $x = 10^{12}$ par `(1+1/x)^x` est très imprécis (on n'obtient que 3 chiffres exacts). Par contre, l'expression est calculée avec plus de 10 chiffres exacts par `exp(x*log1(1/x))`.

```
        print exp(10~^12*log1(10~^-12))
        print (1+10~^-12)^(10~^12)
```

Sortie (215 ms)

```
0.2718281828~ E+1
0.2716110034~ E+1
```

SIN(p)

COS(p)

TAN(p)

ASIN(p)

ACOS(p)

ATN(p)

ATN2(q, p)

V_fonctions Fonctions trigonométriques

p, q

réels

La précision des fonctions `sin`, `cos` et `tan` se détériore pour les grandes valeurs de l'argument, même si l'argument est donné en exact.

Exemple

Le programme suivant écrit l'erreur absolue sur `sin`(p) pour des grandes valeurs de p, de l'ordre de 10^n. En précision 10, cette erreur devient insupportable lorsque p est de l'ordre de 10^{14}.

```
for n=4,20
  print n;abs(sin((10^n+1/6)*pi)-1/2)
next
```

Sortie (1495 ms)

```
...
  10  0.6159473465~ E-5
...
  14  0.8662802676~ E-1
...
```

La fonction `asin`(p), pour $p \in [-1, 1]$, calcule la détermination de l'arc sinus de p comprise dans $[-\pi/2, \pi/2]$.

Exemple

Examine l'erreur relative sur `asin`(`sin`(w)) pour des petits nombres.

```
for i=1,100
  w=(-2)^-i/7~
  y=abs((asin(sin(w))-w)/w)
  h=max(h,y)
next i
print h
```

Sortie (13380 ms)

```
0.3730349363~ E-13
```

La fonction `acos`(p), pour $p \in [-1, 1]$, calcule la détermination de l'arc cosinus de p comprise dans $[0, \pi]$. Au voisinage de $p = 1$, la précision relative correspondant à la précision en cours est obtenue si l'argument est exact. En effet `acos`(p) est calculé à l'aide de `atn2`(q, p) où $q = \sqrt{1-p^2}$ est mieux calculé si p est exact.

Exemple

Le cosinus d'un petit nombre w est calculé avec 50 chiffres, et converti en nombre exact p. Le programme montre que l'erreur relative sur `acos`(p), en précision 10, est inférieure à 10^{-14}. On obtient le même résultat sans convertir `cos`(w) en exact à la troisième ligne du programme. En effet, dans le calcul de `acos`(p), p est alors connu avec 50 chiffres qui sont effectivement utilisés. Si dans ce calcul on remplace `acos`(p) par `acos`(`float`(p)), l'erreur relative est beaucoup plus grande, 0.01 environ.

```
precision 50
w=10^-7
p=exact(cos(w))
precision 10
wp=exact(acos(p))
```

```
        print float(abs(w-wp)/w)
```

Sortie (770 ms)

```
    0.2601726072~ E-14
```

Exemple

La fonction `ang`(a, b, c) détermine l'angle C du triangle ABC à partir des longueurs des côtés. L'exemple écrit les angles d'un triangle rectangle.

```
        a=13
        b=12
        c=5
        print ang(a,b,c);ang(b,c,a);ang(c,a,b)
        stop
    ang:function(a,b,c)
        value=acos((a^2+b^2-c^2)/2/a/b)
        return
```

Sortie (275 ms)

```
    0.3947911197~  0.1570796327~ E+1  0.1176005207~ E+1
```

La fonction `atn`(p) calcule la détermination de l'arc tangente de p comprise dans $[-\pi/2, \pi/2]$. La fonction `atn2`(q, p) renvoie l'argument du nombre complexe $p + iq$. Le résultat est un nombre dans $[-\pi, \pi]$. C'est aussi l'angle que fait le vecteur (p, q) du plan avec l'axe Ox. La forme `atn2`(1, x) donne la détermination de l'arc cotangente de x comprise dans $[0, \pi]$. On a aussi l'égalité `atn2`(y, 1) = `atn`(y).

Exemple

Calcule la somme des angles d'un triangle rectangle de petits côtés a et b.

```
        a=rnd
        b=rnd
        print atn2(a,b)+atn2(b,a)+pi/2
```

Sortie (155 ms)

```
    0.3141592654~ E+1
```

SINH(p)

COSH(p)

TANH(p)

ASINH(p)

ACOSH(p)

ATNH(p)

V_fonctions Fonctions hyperboliques

p

réel

Ces fonctions se distinguent des fonctions circulaires par le suffixe H. Il faut $|p| < 1$ pour `atnh`(p) et $|p| \geq 1$ pour `acosh`(p). Pour ces deux fonctions, l'erreur relative sur le résultat correspondra à la précision en cours, même pour p au voisinage de ± 1 si p est entré en exact.

Exemple

On vérifie les relations $\cosh^2 p - \sinh^2 p = 1$, $\text{atnh}(\tanh p) = p$ ainsi que $\text{acosh}(\cosh p) = p$ pour une valeur aléatoire de p.

```
p=rnd
print cosh(p)^2-sinh(p)^2
print atnh(tanh(p))-p
print acosh(cosh(p))-p
```

Sortie (540 ms)

```
 1.0000000000~
 0.2220446049~ E-14
-0.3552713679~ E-14
```

Exemple

On calcule l'arc sinus hyperbolique pour un petit nombre $p =$ `sinh`(x) d'une part à l'aide de la fonction `asinh`, et d'autre part en utilisant l'expression $\log(p + \sqrt{1+p^2})$. La comparaison avec la valeur théorique x montre que l'erreur relative correspond à la précision en cours pour le calcul par `asinh`, mais pas pour la deuxième méthode.

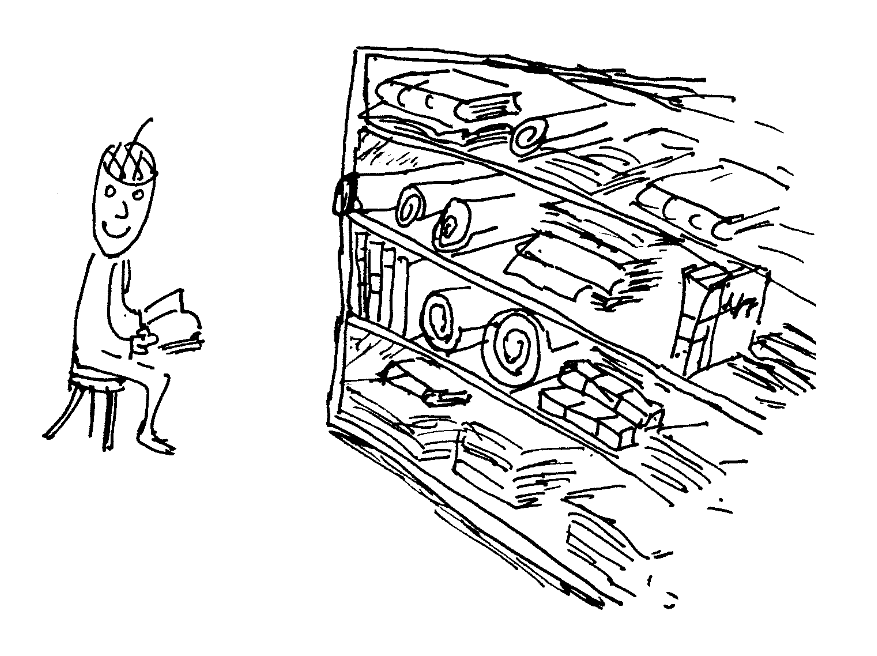
```
x=rnd*10^-6
p=sinh(x)
x1=asinh(p)
x2=log(p+sqr(1+p^2))
print x1;abs(x-x1)/x
print x2;abs(x-x2)/x
```

Sortie (370 ms)

```
0.2898312905~ E-6  0.7306258631~ E-15
0.2898312849~ E-6  0.1935531953~ E-7
```

16

Bibliothèque Mathématique



La bibliothèque MATH.Z contient quelques fonctions et procédures mathématiques d'intérêt général, que nous avons classées en commençant par les méthodes algébriques qui donnent des résultats exacts, sans approximations, et en terminant par les méthodes numériques qui renvoient des approximations (flottants).

Une fonction ou procédure décrite dans ce chapitre peut presque s'utiliser comme si elle faisait partie du Basic, après chargement de la bibliothèque. Les seules différences avec les commandes internes du Basic sont que le mot clef `value` ne peut pas être utilisé dans les arguments et que les noms des programmes doivent être écrits en minuscules, l'option nodistingo n'étant pas utilisée. Ces programmes, qui sont des sources en Basic, peuvent être modifiés et adaptés aux besoins de chacun. Pour en faciliter l'étude et la modification, nous indiquons quels sont les algorithmes utilisés.

Menu B_USER

Des exemples d'utilisation du menu B_USER de la bibliothèque Math ont été donnés dans le chapitre 2. Ce menu permet d'appeler les procédures ci-dessous qui demandent des données à l'aide d'un dialogue. Il vaut mieux effectuer préalablement un `clear` pour éviter des conflits de noms, sauf si on veut entrer des données calculées auparavant et conservées dans des variables.

integre

Procédure Intégration

La procédure `integre` demande une expression, puis écrit son intégrale. Elle utilise la procédure `intg1` qui intègre toute fraction rationnelle.

systeme

Procédure Résolution exacte d'un système d'équations

La procédure `systeme` demande l'entrée d'un système d'équations (qui peut être non linéaire) et le résout. Elle utilise la fonction `sgeq`.

racines

Procédure Racines réelles

La procédure `racines` demande l'entrée d'un polynôme unilittéral et sort ses racines réelles. Elle utilise la procédure `zerop`. Si les racines sont rationnelles, elles sont sorties exactement. Sinon elles sont sorties avec la précision en cours. Ainsi, pour obtenir 100 chiffres exacts, il faut, avant l'appel de `racines` (ou de B_USER), effectuer la commande :

```
precision 100
```

libraryp

Procédure Affiche le nom de la bibliothèque

La procédure `libraryp` affiche la valeur des deux fonctions suivantes `library` et `libraryv`.

library

C_fonction Nom de la bibliothèque

libraryv

V_fonction Numéro de version de la bibliothèque

Exemple

Les instructions suivantes peuvent s'utiliser au début d'un programme d'application pour vérifier que la bibliothèque Math a été chargée.

```
        on error notloaded
        if library<>"MATH"
 notloaded:message "Veuillez charger la| bibliothèque MATH"
          stop
        endif
        on error stop
        'suite
```

Arithmétique

euler_phi(n)

V_fonction Indicateur d'Euler de n

n

entier $n > 0$

La fonction `euler_phi` renvoie le nombre d'entiers $a \in [0, n-1]$ qui sont premiers avec n.

Exemple

On calcule $\varphi(a)$ et $\varphi(b)$ pour deux nombres a et b premiers entre-eux, puis on vérifie les relations :

$$\varphi(ab) = \varphi(a) * \varphi(b)$$

et

$$a^{\varphi(b)} = 1 \bmod b$$

$$b^{\varphi(a)} = 1 \bmod a$$

```
          a=random(2^16)+1
          do
            b=random(2^16)+1
            ift gcdr(a,b)=1 exit
          loop
          ea=euler_phi(a)
          eb=euler_phi(b)
          print a;ea
          print b;eb
          print euler_phi(a*b)-ea*eb;
          print modr(mdpwre(a,eb,b)-1,b);modr(mdpwre(b,ea,a)-1,a)
```

Sortie (325 ms)

```
38184  12096
2801  2800
0  0  0
```

chinoiseq(c1, a1, m1 { , ci, ai, mi })

V_fonction Résolution modulaire

c1, a1, m1, ci, ai, mi

entiers

La fonction résout en x le système d'équations modulaires :

$$c_1 x = a_1 \bmod m_1$$
$$c_2 x = a_2 \bmod m_2$$
$$\cdots$$
$$c_r x = a_r \bmod m_r$$

Les modules m_i doivent être premiers deux à deux (gcdr(m_i,m_j) $= 1$ pour $i \neq j$) et le coefficient c_i de x doit être inversible modulo m_i (gcdr(c_i,m_i) $= 1$). D'après le théorème chinois, il existe alors une solution unique du système modulo $P = m_1 \times m_2 \cdots \times m_r$. La fonction `chinoiseq` renvoie la solution $x \in [0, P[$.

Exemple

On détermine une solution des congruences $19x = 12 \bmod 47$ et $12x = 21 \bmod 121$.

```
print chinoiseq(19,12,47,12,21,121)
```

Sortie (180 ms)

```
1000
```

Méthode

La solution x du système d'équations, lorsque tous les c_i sont égaux à 1, est calculée par `chinois(m, a)` où `m=vset$`(m_1, m_2, ..., m_r) et `a=vset$`(a_1, a_2, ..., a_r). Dans ce calcul, la procédure `chinois1` détermine le v_ensemble

`N=vset$`(N_1, N_2, ..., N_r), où les nombres N_i vérifient les relations $N_i = \delta_{ij} \bmod m_j$. La solution x est alors calculée par `chinois2` comme étant :

$$\sum_{i=1}^{r} N_i a_i \bmod P.$$

legendre(p, q)

V_fonction Symbole de Legendre

p

entier

q

nombre premier $q > 2$

La fonction `legendre` renvoie le symbole de Legendre $(p|q)$. Si $p = 0 \bmod q$, $(p|q) = 0$. Si $p = x^2 \bmod q$ est un résidu quadratique non nul, $(p|q) = 1$. Sinon $(p|q) = -1$.

Exemple

Les nombres 1, 2 et 4 sont les résidus quadratiques, et 3, 5 et 6 les non-résidus modulo 7.

```
print conc$(i=1,6 of i)
print conc$(i=1,6 of legendre(i,7))
```

Sortie (430 ms)

```
 1  2  3  4  5  6
 1  1 -1  1 -1 -1
```

prsqr(a, p)

V_fonction Racine carrée modulaire

a

entier

p

nombre premier

Si a est un résidu quadratique mod p, la fonction `prsqr` renvoie le nombre $x \in [1, p/2]$ tel que $a = x^2 \bmod p$.

Méthode

N Koblitz (1987) p47.

Exemple

On calcule de deux façons différentes une racine carrée de -286 mod 4272943, d'abord par la fonction `prsqr`, puis en factorisant x^2+286 mod 4272943. Les temps des deux calculs, en ms, sont également affichés.

```
clear timer
print prsqr(-286,4272943);" mtimer=";mtimer
clear timer
```

```
print mdff(x^2+286,4272943);" mtimer=";mtimer
```

Sortie

```
1493445 mtimer=  355
 [x +2779498]* [x +1493445] mtimer=  625
```

pollard w

brison w

lenstra w [, p [, a]]

Procédures Factorisation de l'entier w en facteurs premiers

w

entier $w > 1$

p

réel (par défaut $p = \sqrt{w}$)

a

entier*31 (par défaut $a = -1$)

La fonction interne `prfact`(w) convient lorsque le nombre entier w est composé de facteurs premiers $< 10^6$. Les procédures de factorisation introduites ici sont plus rapides que `prfact` pour les grands nombres. La procédure `pollard` convient pour des nombres jusqu'à 25 chiffres, et les procédures `brison` et `lenstra` pour des nombres jusqu'à 30 chiffres. Pour une description générale des méthodes de factorisation, voir D E Knuth (1981 vol 2 chapitre 4.5.4) et N Koblitz (1987 chapitres 5 et 6).

Comparaison des méthodes

N	`prfact`	`pollard`	`brison`	`lenstra`
15	16	34	73	94
17	285	111	93	404
19	2909	456	209	6206
21	35000	1383	296	9953
*21	60877	695	411	8071
23		3606	432	1580
25		18142	1457	?
27		62037	24	4318
29		?	5031	?

La table ci-dessus donne les temps de factorisations (en s) de quelques entiers w (N indique le nombre de chiffres de w). Les nombres w ont été choisis difficiles à factoriser (produit de 2 grands nombres premiers). Le nombre à $N = 2k + 5$ chiffres est $w =$ `prime`$(10^k)\times$`prime`(10^{k+4}). L'exemple noté *21 concerne $2^{67} - 1 = 147573952589676412927$. Le temps 24 s pour le nombre de 27 chiffres n'est

évidemment pas typique. Les ? indiquent que la factorisation n'a pas été trouvée en 24 heures.

pollard

Le résultat de la factorisation est écrit par la procédure. Les facteurs affichés entre parenthèses sont non premiers et non factorisables par la méthode. Les autres facteurs sont probablement premiers.

Méthode

J M Pollard (1975)

La méthode est probabiliste, elle peut échouer (très rarement). Pour trouver un facteur premier a, il faut en général moins de $\sqrt{a}/5$ secondes. Cela donne un temps acceptable pour les facteurs premiers $< 10^{10}$. Le nombre d'itérations est renvoyé dans `pollard_iter`. Les variables/index, sauf `pollard_iter`, utilisés par la procédure sont locaux.

Exemple

Voici la factorisation de $2^{67}-1$ et le nombre d'itérations.

```
pollard 2^67-1
print "timer=";timer;" s  ";pollard_iter;" itérations"
```

Sortie

```
  147573952589676412927=  193707721 * 761838257287
timer=  695 s    13719 itérations
```

brison

La procédure `brison` factorise les grands nombres plus rapidement que `pollard`. Beaucoup de place mémoire est nécessaire, les variables/index utilisés ne sont pas locaux.

Méthode

J Brillhart & M A Morrison (1975)

Exemple

Voici la factorisation de $2^{67}-1$ et le nombre d'itérations. Dans ce cas, la factorisation est plus rapide que par `pollard`.

```
brison 2^67-1
print "timer=";timer;"  nb d'itérations=";iter
```

Sortie

```
  147573952589676412927=  193707721 * 761838257287
timer=  411  nb d'itérations=  6286
```

lenstra

Méthode

H W Lenstra (1987)

La procédure `lenstra` peut être utilisée avec un deuxième argument $p < \sqrt{w}$. Dans ce cas, la méthode recherche plus intensément les facteurs premiers inférieurs à p. Le troisième argument, a, correspond à la courbe elliptique $y^2 =$

$x^3 + ax - a$ utilisée. Si la courbe ne donne pas de factorisation, la recherche d'une factorisation est continuée après incrémentation de a.

Exemple

Le programme suivant factorise (ou du moins essaie de factoriser) les divers entiers qui ont servi à dresser la table de comparaison des méthodes de factorisations.

```
      fact1 prime(10^5)*prime(10^9)
      fact1 prime(10^6)*prime(10^10)
      fact1 prime(10^7)*prime(10^11)
      fact1 prime(10^8)*prime(10^12)
      fact1 2^67-1
      fact1 prime(10^9)*prime(10^13)
      fact1 prime(10^10)*prime(10^14)
      fact1 prime(10^11)*prime(10^15)
      fact1 prime(10^12)*prime(10^16)
      stop
fact1:procedure(x)
      print len(justl$(x));" chiffres  ";
      clear timer
      lenstra x,intsqr(x/2500)
      print "timer=";timer;" s  "
      return
```

Algèbre linéaire

sleq f, z, vz, m

Procédure Résolution d'un système linéaire à $m+1$ équations et inconnues.

m

entier ($m \geq 0$)

z

Les z(i) pour $i = 0, 1, \ldots, m$ doivent être des littéraux (les inconnues). Il s'agit, comme toujours de littéraux au sens généralisé, et pas seulement d'un tableau de type lit. Par exemple le nom z suivant convient :

```
var z(m)
z(0)=x
z(1)=y
```

```
etc.
```
où `x`, `y`, ... sont de type lit.

f

Les $m+1$ équations sont données par `f`$(i) = 0$ (pour $i = 0, 1, \ldots, m$). Par exemple, pour résoudre :

$$\begin{cases} x + y = 3 \\ x + 7y = 9 \end{cases}$$

on initialisera un tableau `f` par :

```
var f(1)
f(0)=x+y-3
f(1)=x+7*y-9
```

vz

Le nom `vz` doit être un tableau de variables défini par (`M`$\geq m$) :

```
var vz(M)
```

La procédure `sleq` renvoie dans le tableau `vz` la solution du système d'équations. La valeur de l'inconnue `z`(i) est donnée par `vz`(i) (pour $i = 0, 1, \ldots, m$). La procédure n'admet que les systèmes ayant une solution et une seule. Pour des systèmes linéaires singuliers, utiliser les programmes de résolution généraux `sgeq`, `sgeqd` et `sgeqe`. Les coefficients des équations ne sont pas limités aux nombres, ils peuvent contenir des littéraux.

Ne pas utiliser de noms commençant par `sleq` pour arguments de `sleq`, sinon il peut y avoir des conflits avec les variables locales de la procédure.

Méthode

Résolution exacte par élimination en utilisant la fonction interne du Basic `elim`.

Exemple

Résolution en z_i $(i = 0, 3)$ du système d'équations linéaires :

$$\begin{cases} (b-1)(z_1+z_2+z_3) = a+3b-3 \\ 2z_2+3z_3 = 10 \\ 4z_2+9z_3+16z_0 = 48 \\ 8z_2+27z_3+64z_0 = 164 \end{cases}$$

```
lit z(3)
var f(3),vz(3)
f(0)=z(3)*b -z(3) +z(2)*b -z(2) +z(1)*b -z(1) -a -3*b +
 3
f(1)=3*z(3) +2*z(2) -10
f(2)=9*z(3) +4*z(2) +16*z(0) -48
f(3)=27*z(3) +8*z(2) +64*z(0) -164
sleq f,z,vz,3
```

```
        print "z(0)=";vz(0)
        print "z(1)=";vz(1)
        print "z(2)=";vz(2)
        print "z(3)=";vz(3)
```

Sortie (1380 ms)

```
z(0)=  1
z(1)=   [a]* [b -1]^-1
z(2)= -1
z(3)=  4
```

invm M, R, m

Procédure Inversion de la matrice M d'ordre $m+1$

m

entier ($m \geq 0$)

M

`M`$(i,\ j)$ pour i et j entiers $\in [0,m]$ doit renvoyer l'élément M_{ij} de la matrice à inverser. Chaque élément est appelé une fois et une seule par `invm`, dans l'ordre M_{00}, M_{01}, …, M_{0m}, M_{10}, M_{11}, …, M_{mm}. Le nom `M` peut être le nom d'un tableau à 2 indices ou d'une fonction.

R

Le nom `R` doit être défini par (`A` $\geq m$ et `B` $\geq m$) :

```
    var R(A,B)
```

Après l'appel de la procédure `invm`, `R`$(i,\ j)$ pour i et j entiers $\in [0,m]$ représente l'inverse R de la matrice d'entrée M. Il est possible d'utiliser le même tableau pour l'entrée et la sortie. Les éléments de la matrice M doivent être des expr exactes (ils peuvent contenir des littéraux). Pour inverser une matrice flottante, il faut d'abord la convertir en exact. L'inversion est faite exactement, sans approximation.

Ne pas utiliser de noms commençant par `invm` pour arguments de `invm`, sinon il peut y avoir des conflits avec les variables locales de `invm`.

Méthode

Résolution par `sleq` de l'équation matricielle $Mz = w$.

Exemple

Le programme suivant inverse exactement la matrice de Hilbert d'ordre $n = D+1$ dont l'élément $(i,\ j)$ (i et j vont de 0 à D) est $1/(i+j+1)$. La table suivante donne les temps d'inversion (en s) pour divers n.

n	5	10	20	30	40
temps	2	20	271	1413	5362

```
    D=4
```

```
print "Inversion de la matrice de Hilbert";D+1;"  X";D+
 1;" :"
var M(D,D),N(D,D)
for i=0,D
  for j=0,D
    M(i,j)=1/(i+j+1)
    N(i,j)=M(i,j)
    print justr$(M(i,j),8);
  next j
  print
next i
clear timer
invm M,M,D
print "Inverse (calculé en";mtimer;" ms) :"
MX=0
for i=0,D
  for j=0,D
    MX=max(MX,abs(M(i,j)))
    print justr$(M(i,j),8);
  next j
  print
next i
print "Elément max=";MX
print "Réinversion et vérification"
clear timer
invm M,M,D
print timer
for i=0,D
  for j=0,D
    ift N(i,j)<>M(i,j) print "erreur",i,j
next j,i
```

Sortie

```
Inverse (calculé en  2560 ms):
      25    -300    1050   -1400     630
    -300    4800  -18900   26880  -12600
    1050  -18900   79380 -117600   56700
   -1400   26880 -117600  179200  -88200
     630  -12600   56700  -88200   44100
```

polyappr(f, D, x, R)

V_fonction Approximation polynomiale

x

littéral

D

entier $D \geq 0$

f

nom de fonction

La fonction $f(x)$ sera appelée par `polyappr` en $D+2$ points de l'intervalle $[0,1]$ avec un argument flottant.

R

nomi de type var

La fonction `polyappr` renvoie un polynôme $w(x)$ de degré D, approchant la fonction $f(x)$ sur $[0,1]$. La variable `R` contient en sortie une estimation de l'erreur sur $[0,1]$ (en flottant).

Méthode

La méthode est basée sur le théorème d'alternance de Tchebycheff : Il existe $D+2$ points Y_i (pour $i = 1, 2, \ldots, D+2$) dans $[0,1]$ et une constante R tels que :

$$R = (-1)^i |w(Y_i) - f(Y_i)| \qquad \text{(s)}$$

$$|w(x) - f(x)| \leq |R| \;\; \text{pour tout } x \in [0,1]$$

si et seulement si $w(x)$ est le polynôme de degré D qui minimise $|w-f|$ sur $[0,1]$.

Le programme `polyappr` résout le système (s) pour les inconnues R et les coefficients du polynôme w, pour des points Y_i donnés choisis par `polyappr`. L'approximation pourrait être améliorée en optimisant le choix de ces points Y_i. Ce programme a été utilisé pour obtenir les polynômes approchant les fonctions `exp`, `log`, `tan`, `atn` du Basic 1000d (lorsque la précision est 10 ou moins).

Exemple

Calcul d'une approximation de $\text{EXX}(x) = e^x + \sin x$ ($x \in [0,1]$). La fonction EXX est approchée par un polynôme $w(x)$ de degré 3, puis l'erreur est déterminée par calcul de $|\text{EXX}(x) - w(x)|$ pour des valeurs x aléatoires (arrêt par Break).

```
print "Calcul d'une approximation de EXX(x)=exp(x)+sin(
 x) sur [0,1]"
w=polyappr(EXX,3,x,R)
print "  timer=";timer;". polynôme approché à";R;" près
 "
print "w=";w
print "Détermination aléatoire de l'erreur"
np=0
do
  S=rnd
  S=abs(EXX(S)-fsubs(w,x=S))
  np=np+1
  if S>R
```

```
            R=S
            print "Erreur >~";justl$(R,20);"(Vérifié en";np;" p
             oints)"
          endif
        loop
        stop
    EXX:function
        value=exp(@1)+sin(@1)
        return
```

Sortie

```
  timer=  10. polynôme approché à  0.6975591747~ E-3 près
w=  284006/2081625*x^3 +894791/2224375*x^2 +6079245/3007961*x +6168647
 /6172953
Détermination aléatoire de l'erreur
...
Erreur >~0.7024234232~ E-3  (Vérifié en  15603 points)
```

Système d'équations

La méthode de résolution d'un système d'équations est basée sur l'utilisation de la fonction `elim`(eq1, eq2, x) qui élimine x entre eq1 et eq2, et de la fonction `formf` qui factorise les expressions littérales. Les équations peuvent être non linéaires. Même dans ce cas le programme trouve toutes les solutions rationnelles, et exprime les solutions non rationnelles en termes de zéros de polynômes à une inconnue. Les équations peuvent comporter d'autres littéraux que les inconnues, les solutions obtenues sont alors les solutions génériques, valables pour toutes les valeurs de ces autres littéraux.

Les nombres d'équations et d'inconnues peuvent être différents. Les équations peuvent ne pas être indépendantes, pour chaque relation entre équations on obtient le message "équations non indépendantes". S'il y a moins d'équations indépendantes que d'inconnues le programme exprime les solutions en fonction d'inconnues pouvant être arbitraires. S'il y en a plus on obtient le message "système impossible". Si au cours de la résolution il y a un facteur commun entre 2 équations, la résolution est faite d'abord en supposant ce facteur différent de zéro, puis ensuite en remplaçant les deux équations par ce facteur.

sgeq(m, n, eq, z)

sgeqd(m, n, eq0, eq1, ..., eqm, z0, z1, ..., zn)

sgeqe(m, n, eq0, eq1, ..., eqm, z0, z1, ..., zn)

C_fonctions Résolution de $m+1$ équations à $n+1$ inconnues

m, n

entier ($m \geq 0$, $n \geq 0$)

eq, z

Noms de formes indicées, eq(i) et z(j) avec $i = 0, 1, \ldots, m$ et $j = 0, 1, \ldots, n$. Le nom eq peut être un tableau de type var ou un nom de fonction. Le nom z peut être un tableau de type var ou lit, ou un nom de fonction.

eq1, ..., eqm, z1, ..., zn

expr

Les formes non indicées eq1, ..., eqm, ou bien les valeurs eq(0), ..., eq(m) donnent les $m+1$ équations :

```
eq(0)=0
eq(1)=0
...
eq(m)=0
```

Seul le numérateur, `num`(eq(i)), est pris en compte. Les formes z0, ..., zn ou z(0), ..., z(n) sont des littéraux qui représentent les $n+1$ inconnues du système.

Les trois C_fonctions `sgeq`, `sgeqd` et `sgeqe` écrivent la solution du système d'équations et renvoient une chaîne contenant ce qui a été écrit. Cela permet de conserver, pour la relire tranquillement, la solution, qui peut prendre plusieurs pages écran. La différence entre `sgeqd` et `sgeq` est seulement dans l'entrée (indicée pour `sgeq` et développé pour `sgeqd`). La fonction `sgeqe` écrit de plus les équations de départ.

Exemple

Résolution de l'équation en x :

$$1989(b+1)x^7 - (b+5968)x^6 + 3x^5 - 3978(b+1)x^2 + (2b+11936)x - 6 = 0$$

```
c$=sgeqd(0,0,1989*(b+1)*x^7-(b+5968)*x^6+3*x^5-3978*(b+
 1)*x^2+(2*b+11936)*x-6,x)
```

Sortie (2510 ms)

```
  3 cas pour  x
Cas  1 pour  x
  x  est un zéro de  x^5 -2
Cas  2 pour  x
  x=  1/1989
Cas  3 pour  x
  x=  3* [b +1]^-1
```

Exemple

Résolution en x et y du système d'équations :

$$\begin{cases} x + y = 1 \\ x^3 + a^2 y = a^2 \end{cases}$$

```
var eq(1),z(1)
eq(0)=x+y-1
eq(1)=x^3+a^2*(y-1)
z(0)=x
z(1)=y
c$=sgeq(1,1,eq,z)
```

Sortie (1870 ms)

```
   3 cas pour  x
 Cas  1 pour  x
   x=  0
   y=  1
 Cas  2 pour  x
   x=  a
   y= -a +1
 Cas  3 pour  x
   x= -a
   y=  a +1
```

Exemple

Résolution en x et y du système d'équations :

$$\begin{cases} 2x^2 - xy - 3x - y^2 + 3y = 0 \\ x^2 + 2xy - 7x - 3y^2 + 7y = 0 \end{cases}$$

```
c$=sgeqe(1,1,2*x^2 -x*y -3*x -y^2 +3*y,  x^2 +2*x*y -7*
 x -3*y^2 +7*y, x, y)
```

Sortie (1885 ms)

```
 Résolution en  x  y du système
   2*x^2 -x*y -3*x -y^2 +3*y= 0
   x^2 +2*x*y -7*x -3*y^2 +7*y= 0
 Solution
 Facteur commun F1=  x -y
 1.a. Cas F1<>0
   x=  2/5
   y=  11/5
 1.b. Cas F1=0
 L'inconnue  x est arbitraire
   y=  x
```

Exemple

Si on désire trouver à quelles conditions sur des littéraux supplémentaires il peut y avoir des solutions, il faut également déclarer certains de ces littéraux comme inconnues. Les relations recherchées apparaissent comme des solutions. Dans cet exemple on montre comment trouver la condition sur a, b et c pour que l'équation $w = ax^2 + bx + c = 0$ ait une racine double. On résout le système :

$$\begin{cases} w = 0 \\ \dfrac{dw}{dx} = 0 \end{cases}$$

en b et x. La condition recherchée, $4ac - b^2 = 0$, apparaît comme solution de b.

```
w=a*x^2+b*x+c
c$=sgeqe(1,1, w, der(w,x),b,x)
```

Sortie (1245 ms)

```
Résolution en  b  x du système
  a*x^2 +x*b +c= 0
  2*a*x +b= 0
Solution
  b  est un zéro de  4*a*c -b^2
  x= -1/2* [b]* [a]^-1
```

Polynômes symétriques

Les fonctions suivantes traitent les polynômes symétriques suivant les n littéraux x(1), x(2), ..., x(n). On utilise aussi un littéral Z d'homogénéisation.

symsigma(k, n, x)

V_fonction symétrique σ_k

symsum(k, n, x)

V_fonction symétrique S_k

k, n

entiers

x

x(i) pour $i = 1, 2, \ldots, n$ doit être un littéral (noté x_i)

La fonction `symsigma` renvoie $\sigma_k = \sum_{i=1}^{n} x_i^k$. La fonction `symsum` renvoie la somme symétrique :

$$S_k = \sum_{i_1 < i_2 < \cdots < i_k} x_{i_1} \times x_{i_2} \times \cdots \times x_{i_k}$$

Exemple

```
var u(3)
u(1)=x
u(2)=y
u(3)=z
print symsigma(2,3,u)
print symsum(2,3,u)
```

Sortie (770 ms)

```
x^2 +y^2 +z^2
x*y +x*z +y*z
```

symf(w, n, x, s, sv [, Z [, *]])

V_fonction Récrit w en fonction des polynômes symétriques sv(i)

n

entier

x

x(i) pour $i = 1, 2, \ldots, n$ doit être un littéral (noté x_i)

w

poly

Si le septième argument est omis, w doit être un polynôme symétrique en x_1, …, x_n. Si le septième argument existe (peu importe sa valeur), w sera d'abord remplacé par son symétrisé en x_1, …, x_n (à un facteur près).

Z

littéral

Si Z est donné, w sera homogénéisé par Z.

s

s(i) pour $i = 1, 2, \ldots, n$ doit être un littéral (noté s_i). Ces littéraux servent à écrire la valeur renvoyée par `symf`.

sv

sv(i) pour $i = 1, 2, \ldots, n$ doit être un polynôme symétrique en x_1, …, x_n de degré i. Le littéral s_i représente la valeur sv(i). Le plus souvent on utilise pour sv les polynômes symétriques σ_i ou S_i. Dans ces deux cas, il suffit d'initialiser un tableau sv avec des valeurs calculées par les fonctions `symsigma` ou `symsum`.

La fonction `symf` renvoie l'expression de w en fonction des polynômes symétriques s_i.

Exemple

La première sortie montre que pour $n = 4$:

$$S_2 = \sum_{i<j} x_i x_j = \frac{\sigma_1^2 - \sigma_2}{2}.$$

La deuxième sortie qui montre la même égalité à un facteur près est obtenue à partir de l'entrée x_1x_2 que la fonction `symf` se charge de symétriser (présence du septième argument). La dernière sortie montre que pour $n = 4$:

$$\sigma_5 = \sum_i x_i^5 = -5S_4S_1 - 5S_3S_2 + 5S_3S_1^2 + 5S_2^2S_1 - 5S_2S_1^3 + S_1^5.$$

```
n=4
lit x(n),S(n),Z
var sv1(n),sv2(n)
for i=1,n
  sv1(i)=symsigma(i,n,x)
next i
for i=1,n
  sv2(i)=symsum(i,n,x)
next i
print symf(sv2(2),n,x,S,sv1,Z)
print symf(x(1)*x(2),n,x,S,sv1,Z,*)
WP=sum(i=1,n of x(i)^5)
print symf(WP,n,x,S,sv2,Z)
```

Sortie (7740 ms)

```
-1/2*S(2) +1/2*S(1)^2
-2*S(2) +2*S(1)^2
-5*S(4)*S(1) -5*S(3)*S(2) +5*S(3)*S(1)^2 +5*S(2)^2*S(1) -5*S(2)*S(1)^
3 +S(1)^5
```

Expressions trigonométriques

Nous montrons ailleurs (`gcd1`) comment le Basic 1000d peut être facilement employé pour traiter les nombres algébriques, par exemple $\sqrt{3}$, de façon exacte. Dans ce chapitre, nous allons voir que le Basic 1000d peut également être programmé pour traiter exactement des expressions mathématiques faisant intervenir des fonctions transcendantes. Les fonctions étudiées ici acceptent en effet des expressions exactes contenant les fonctions trigonométriques `sin`, `cos`, `tg` et `cotg`. Voici des exemples :

```
(sin(x^2+3)+cos(a*tg(u-8)-b*cotg(y)))/(1+h^2-sin(r2))
sin(x+5)^4-cos(x+5)^4+2*cos(x+5)^2
sin(2+x)+cos(cos(x+2))
```

Pour traiter une telle expression, nous la remplaçons par un ensemble de fractions rationnelles, permettant de la décrire, en introduisant des littéraux

supplémentaires qui représentent des `sin` et `cos`. Ces littéraux ont pour noms ici `trigo_l`(0,i) pour les sinus et `trigo_l`(1,i) pour les cosinus. Détaillons la dernière expression ci-dessus. Nous posons :

```
e1=x+2
e2=trigo_l(1,0)
e3=trigo_l(0,0)+trigo_l(1,1)
```

L'expression de départ est entièrement décrite par ces trois expressions rationnelles, car on peut la retrouver comme étant égale à e3, après les substitutions suivantes :

```
trigo_l(1,1) →  cos(e2)
trigo_l(0,0) →  sin(e1)
trigo_l(1,0) →  cos(e1)
```

Dans les programmes ci-après, l'expression sera codée par le v_ensemble `vset$`(e1, e2, e3). Notre cher lecteur aura sans doute remarqué dans cet exemple que le programme de codage reconnaît que c'est le même argument e1 qui apparaît en 2 endroits différents, une fois comme argument de `sin` et une autre fois comme argument de `cos`. Les programmes tiennent compte aussi des relations fonctionnelles :

```
tg(x)=sin(x)/cos(x)
cotg(x)=cos(x)/sin(x)
sin(x)^2+cos(x)^2=1
```

de sorte que la deuxième expression donnée en exemple est reconnue comme étant égale à 1.

Les programmes fournis permettent seulement les simplifications indiquées et le calcul des dérivées des expressions trigonométriques. En utilisant les mêmes méthodes, le lecteur pourra écrire des programmes plus intelligents que les notres, par exemple en leur faisant reconnaître le nombre π (traité de façon exacte), les fonctions trigonométriques inverses, les fonctions `exp` et `log`, en introduisant d'autres relations fonctionnelles, etc.

Pour éviter des conflits de noms, ne pas utiliser de noms commençant par `trigo_` dans les expressions trigonométriques.

trigox(c)

C_fonction Codage de l'expression trigonométrique c

c

exprchaîne, contient l'écriture usuelle

La fonction `trigox` renvoie le v_ensemble décrivant l'expression.

trigop(t)

C_fonction Ecriture de l'expression trigonométrique t

t

v_ensemble décrivant l'expression

La fonction `trigop` renvoie la chaîne alphanumérique représentant l'expression. L'écriture est faite seulement en termes de `sin` et `cos`. Les deux fonctions

`trigox` et `trigop` sont inverses l'une de l'autre. On peut simplifier les expressions trigonométriques comme dans l'exemple suivant :

Exemple

Le programme montre les relations :

$$2\cos^2 x + \sin^2 x = \cos^2 x + 1$$

$$\sin^4(x+5) - \cos^4(x+5) + 2\cos^2(x+5) = 1$$

```
print trigop(trigox("2*cos(x)^2+sin(x)^2"))
print trigop(trigox("sin(x+5)^4-cos(x+5)^4+2*cos(x+5)^2
 "))
```

Sortie (1465 ms)

```
cos( x )^2 +1
1
```

dertrigo(t, x)

C_fonction Dérivée d'une expression trigonométrique

x

littéral

t

v_ensemble décrivant l'expression.

La fonction `dertrigo` dérive, par rapport au littéral x, l'expression trigonométrique codée par le v_ensemble t. Elle renvoie le v_ensemble qui code la dérivée.

Exemple

Le programme affiche la dérivée de $(x-2)\sin(2x+1)$, puis des expressions entrées au clavier.

```
char c
c="sin(2*x+1)*(x-2)"
do
  c=trigox(c)
  print "La dérivée en x de  ";trigop(c)
  print "est ";trigop(dertrigo(c,x))
  input "Entrer une expression trigonométrique",c
loop
```

Sortie (exemple de dialogue)

```
La dérivée en x de  sin( 2*x +1 )*x -2*sin( 2*x +1 )
est 2*cos( 2*x +1 )*x -4*cos( 2*x +1 ) +sin( 2*x +1 )
Entrer une expression trigonométrique
cos(x^2+1)
La dérivée en x de  cos( x^2 +1 )
est -2*sin( x^2 +1 )*x
```

Algèbre de Racah

La fonction `quacg` calcule la valeur exacte des coefficients de Clebsch-Gordan du groupe SU(2). Les fonctions `qua3j`, `qua6j` et `qua9j` renvoient les valeurs exactes des coefficients 3j, 6j et 9j. Les arguments j_1, j_2, ..., m_1, m_2, ... de ces fonctions sont des entiers ou demi-entiers (par exemple 3/2).

Les carrés des valeurs de ces coefficients sont rationnels. Pour représenter exactement une telle valeur, w, nous utilisons sa forme QUA, q qui est le nombre rationnel $q = \mathrm{sgn}(w)w^2$. Par exemple $w = -\frac{1}{5}\sqrt{\frac{2}{3}}$ a pour forme QUA $q = -2/75$. La C_fonction `quac`(q) permet d'écrire la valeur exacte du nombre w à partir de sa forme QUA q, et la fonction `quaf`(q) sa valeur approchée flottante. La fonction `quasum` permet de sommer exactement des nombres à partir de leur forme QUA.

Méthode

Les coefficients 3j sont calculés par la formule de l'appendice 1 du livre de A P Jucys & A A Bandzaitis (1965). Les coefficients 6j sont calculés par la formule de Racah et les 9j comme somme d'un produit de trois 6j (eq (22.6) et eq (24.33) ibid.).

Temps de calcul

Pour divers j, voici les temps de calcul (en s) des 3j, 6j et 9j :

$$\begin{pmatrix} j & j & j \\ 0 & 0 & 0 \end{pmatrix} \qquad \begin{Bmatrix} j & j & j \\ j & j & j \end{Bmatrix} \qquad \begin{Bmatrix} j & j & j \\ j & j & j \\ j & j & j \end{Bmatrix}$$

j	3j	6j	9j
0	.1	.2	1
10	.4	.8	47
20	.8	1.6	167
30	1.1	2.5	392
90	5.3	15.7	5879

quac(q)

C_fonction Valeur exacte de la forme QUA q

quaf(q)

V_fonction Valeur flottante la forme QUA q

q

réel exact (forme QUA)

La fonction `quac` renvoie une chaîne contenant la décomposition du nombre en produit $\prod_{i=1}^{n} p_i^{e_i}$ où les p_i sont des nombres premiers s'ils sont inférieurs à 10^8 et les e_i sont des entiers ou demi-entiers. La fonction `quaf` renvoie la valeur flottante $\mathrm{sgn}(q)\sqrt{|q|}$ de la forme QUA q.

Exemple

La forme QUA $q = -2/75$ représente $-\frac{1}{5}\sqrt{\frac{2}{3}} \approx -0.1632993162$

```
print quac(-2/75);"=";quaf(-2/75)
```

Sortie (800 ms)

```
-  2^(1/2) *  3^(-1/2) *  5^-1= -0.1632993162~
```

quasum({ qi })

V_fonction Somme de formes QUA

qi

réel exact (forme QUA)

Si $q_1, \ldots, q_n$ sont les formes QUA des nombres $w_1, \ldots, w_n$, alors la fonction `quasum` renvoie la forme QUA de $w_1 + \cdots + w_n$. Si cette somme n'est pas de carré rationnel, sortie erreur.

Avec cette fonction, on peut calculer des expressions contenant les opérations $+$ $-$ $\times$ et $/$ à partir des formes QUA. Par exemple la forme QUA de $w_1 \times w_2 - w_3/w_4 + 7w_5$ s'obtient par `quasum`$(q_1 \times q_2, -q_3/q_4, 7^2 q_5)$.

Exemple

L'instruction montre que $\sqrt{2} - 1/\sqrt{2} = 1/\sqrt{2}$

```
print "La somme de";quac(2);"  et de";quac(-1/2);  "
 est";quac(quasum(2,-1/2))
```

Sortie (925 ms)

```
La somme de    2^(1/2)  et de -  2^(-1/2)   est    2^(-1/2)
```

quacg(j1, m1, j2, m2, j, m)

qua3j(j1, j2, j3, m1, m2, m3)

qua6j(j1, j2, j3, j4, j5, j6)

qua9j(j1, j2, j3, j4, j5, j6, j7, j8, j9)

V_fonctions Coefficients Clebsch-Gordan, 3j, 6j et 9j

qua3jp j1, j2, j3, m1, m2, m3

qua6jp j1, j2, j3, j4, j5, j6

qua9jp j1, j2, j3, j4, j5, j6, j7, j8, j9

Procédures Coefficients 3j, 6j et 9j

j1, m1, j2, m2, ...

entier ou demi-entier

Les fonctions `quacg`, `qua3j`, `qua6j` et `qua9j` renvoient les valeurs sous la forme QUA des coefficients suivants :

$$\texttt{quacg}(j_1, m_1, j_2, m_2, j, m) = (j_1, m_1, j_2, m_2 | j, m)$$

$$\texttt{qua3j}(j_1, j_2, j_3, m_1, m_2, m_3) = \begin{pmatrix} j_1 & j_2 & j_3 \\ m_1 & m_2 & m_3 \end{pmatrix}$$

$$\texttt{qua6j}(j_1, j_2, j_3, j_4, j_5, j_6) = \begin{Bmatrix} j_1 & j_2 & j_3 \\ j_4 & j_5 & j_6 \end{Bmatrix}$$

$$\texttt{qua9j}(j_1, j_2, j_3, j_4, j_5, j_6, j_7, j_8, j_9) = \begin{Bmatrix} j_1 & j_2 & j_3 \\ j_4 & j_5 & j_6 \\ j_7 & j_8 & j_9 \end{Bmatrix}$$

Les procédures `qua3jp`, `qua6jp` et `qua9jp` écrivent les valeurs des coefficients 3j, 6j et 9j.

Exemple

Calcule les valeurs :

$$\begin{pmatrix} 7 & 5 & 7 \\ 4 & 2 & -6 \end{pmatrix} = \sqrt{\frac{11}{2 \times 3 \times 17 \times 19}}$$

$$\begin{Bmatrix} 2 & 5/2 & 7/2 \\ 3/2 & 3 & 3 \end{Bmatrix} = -\frac{13}{2^2 \times 7}\sqrt{\frac{1}{2 \times 3 \times 5}}$$

$$\begin{Bmatrix} 2 & 2 & 2 \\ 3/2 & 3/2 & 3/2 \end{Bmatrix} = 0$$

$$\begin{Bmatrix} 5 & 3 & 4 \\ 2 & 4 & 3 \\ 3 & 2 & 1 \end{Bmatrix} = \frac{1}{3^2 \times 7^2}\sqrt{\frac{13}{2 \times 5}}$$

```
qua3jp 7,5,7,4,2,-6
qua6jp 2,5/2,7/2,3/2,3,3
qua6jp 2,2,2,3/2,3/2,3/2
qua9jp 5,3,4,2,4,3,3,2,1
```

Sortie (4640 ms)

```
3j(  7  5  7  4  2 -6)=    2^(-1/2) *  3^(-1/2) *  11^(1/2) *  17^(-1/
  2) *  19^(-1/2) =  0.7533893145~ E-1
6j(  2  5/2  7/2  3/2  3  3)= -  2^(-5/2) *  3^(-1/2) *  5^(-1/2) *  7
  ^-1 *  13 = -0.8476658628~ E-1
6j(  2  2  2  3/2  3/2  3/2)=  0 =  0~
9j(  5  3  4  2  4  3  3  2  1)=    2^(-1/2) *  3^-2 *  5^(-1/2) *  7^
  -2 *  13^(1/2) =  0.2585431803~ E-2
```

Intégration algébrique

La fonction interne du Basic, `intg` ne peut intégrer que les fractions dont le dénominateur se factorise en polynômes de degré 1. Voici des procédures et fonctions qui permettent l'intégration toute fraction rationnelle. L'intégrale F d'une fraction rationnelle f quelconque est la somme d'une fraction rationnelle r et d'une combinaison linéaire S de logarithmes :

$$S = \sum_{i=1,n} y_i \log b_i.$$

Les y_i sont des nombres algébriques, et le polynôme b_i est rationnel en y_i. Les y_i seront définis comme étant les zéros d'un polynôme. La somme S renvoyée ici est telle que le degré de l'extension algébrique sur $\mathbf{Q}$ engendrée par les y_i soit minimum.

intg1 f, x

Procédure Intègre f en x et écrit le résultat

f

expr

x

littéral

La procédure `intg1` convient si on désire seulement l'impression.

Exemple

Calcul de l'intégrale de w. Par la fonction interne `intg`, ce calcul n'est pas possible.

```
w=der(1/(x^2+a*x+b),x) +1/(x-7)+8/(x+7)
print w
intg1 w,x
```

Sortie (6455 ms)

```
 [la fonction à intégrer s'écrit sur trois lignes]
La partie rationnelle de l'intégrale est
                  [x^2 +x*a +b]^-1
La partie logarithmique de l'intégrale est la somme de
  8  *  log(  x +7  )
et de
  log(  x -7  )
```

Les procédures suivantes permettent le traitement de l'intégrale.

intg2 f, x, reg, nreg

Procédure Intégration de f suivant x (partie rationnelle)

f

expr

x

littéral

reg, nreg

nomi de type var (contiennent le résultat)

L'intégrale $\int f\,dx = r + S$ est la somme d'une fraction rationnelle r et d'une somme S de logarithmes de polynômes. La procédure `intg2` renvoie r dans la variable reg, et $\frac{dS}{dx}$ dans la variable nreg. Pour obtenir S à partir de nreg, on utilisera `intg3` ou `intg4`.

intg3 nreg, x, y, R, b

Procédure Intégration de la partie logarithmique nreg en x

nreg

expr

x

littéral

y

littéral utilisé pour renvoyer l'intégrale (il faut $y \neq x$)

R, b

nomi de type var

Ces variables, contenant le résultat, doivent être déclarées par :

```
var R,b(N)
```

où `N` est suffisamment grand. `N` = **deg**(**den**(nreg),x) + 1 est toujours suffisant. L'expr nreg doit être une sortie de `intg2`, et non nul. L'intégrale de nreg est calculée par `intg3` sous la forme :

$$b_1 \times \sum_{i=2}^{n} \sum_{j=1}^{k_i} y_{i,j} \log B_{i,j}.$$

n est un entier renvoyé dans b(0). C'est aussi `factorn`(R), le nombre de facteurs de R. b_1 est un facteur multiplicatif de l'intégrale qui est renvoyé dans b(1). Le i-ième facteur de R, `factorp`(R,i) est un polynôme de degré k_i en y. Il a exactement k_i zéros distincts $y_{i,j}$ pour $j = 1, 2, \ldots, k_i$. Le nombre $y_{i,j}$ est rationnel si et seulement si $k_i = 1$. Ces $y_{i,j}$ sont les coefficients des logarithmes de l'intégrale. L' expression $B_{i,j}$ dans le logarithme s'obtient à partir du polynôme en y renvoyé dans b(i), qui est de degré $< k_i$, en substituant y par $y_{i,j}$.

intg4(nreg, x)

C_Fonction Intégrale de la partie logarithmique nreg en x

nreg

expr

x

littéral

La fonction `intg4` renvoie une chaîne dans laquelle est écrit le résultat de l'intégration.

gcd1(A, B, C, x, y)

V_fonction Pgcd dans une extension algébrique

x, y

littéraux

A, B

poly (en x, y et autres littéraux)

C

poly en y de degré $k > 0$

Le polynôme C peut contenir d'autres littéraux mais doit être indépendant de x (`deg`(C, x)=0).

La fonction `gcd1` est utilisée par `intg3`, mais présente un intérêt général. C'est un exemple de calculs en nombres algébriques. Si on veut traiter exactement $\sqrt{2}$, on utilise un littéral y, représentant ce nombre, et on introduit le polynôme $C = y^2 - 2$. Les diverses expressions contenant y peuvent alors se simplifier en prenant leur reste modulo C. Pour les multiplications, additions, etc., il suffit d'effectuer des calculs conditionnels :

```
cond y^2-2
```

La fonction `gcd1` permet de calculer le pgcd des deux polynômes A et B, contenant un nombre algébrique, représenté par le littéral y et défini par la condition $C = 0$. Elle renvoie un polynôme de degré en y inférieur à k. De plus, si c'est possible (ce qui est le cas si x et y sont les seuls littéraux et si C est irréductible), son coefficient de plus haut degré en x est rendu indépendant de y.

Sommation algébrique

La fonction `dsum`(f, x, a, b) permet de sommer sur l'entier $x \in [a, b]$ (a et b désignent des entiers $a < b$) certaines expressions rationnelles f. Comme a et b peuvent contenir des littéraux, cette fonction est très différente de `sum`($x = a, b$ of f). La méthode consiste à déterminer une fonction $F(x)$ telle que :

$$F(x) - F(x-1) = f(x) \qquad (1)$$

La somme vaut alors $F(b) - F(a-1)$. Si $f(x)$ est un polynôme de degré k en x, $F(x)$ est un polynôme de degré $k+1$. La fonction `sdpoly` le calcule par

simple identification dans l'équation (1). Si $f(x)$ est une fraction rationnelle en x l'algorithme de Gosper (1978) est utilisé. Cet algorithme permet d'obtenir $F(x)$ si $F(x)/F(x-1)$ est rationnel en x. Il est programmé dans la fonction `sdrap`.

Comme exemple de fonctions $f(x)$ acceptables, citons :

$$f(x) = \frac{N(x)}{(x+n_1)\times(x+n_2)\times\cdots\times(x+n_k)}$$

où $N(x)$ est un polynôme en x et n_1, n_2, ..., n_k sont des nombres entiers naturels différents.

Temps de calculs

Voici les temps de calcul (en s) pour $f(x) =$ `ppwr`(x, k)

k	temps	k	temps
0	0.2		
2	0.3	−2	0.5
5	1.5	−5	1.7
10	4	−10	6
20	22	−20	35
50	500		

dsum(f, x, a, b)

V_fonction Sommation algébrique $\sum_{x=a}^{b} f(x)$

f, a, b

expr

x

littéral

La fonction `dsum` renvoie la somme de $f(x)$ sur les entiers $x \in [a, b]$, si cette somme est rationnelle. Sinon message et arrêt. La fonction $f(x)$ est donnée par l'expr `f` contenant le littéral x.

Exemple

La première sortie montre que la somme sur x des entiers de 1 à n est $(n^2+n)/2$.

```
        printdsum x
        printdsum x^10
        printdsum 1/x/(x+2)/(x+8)
        bary (2*j+1)*(j*(j+1)-s*(s+1)-l*(l+1))
        stop
printdsum:print "La somme sur les valeurs entières x de 1 à n de
 @1f est"
```

```
          print dsum(@1,x,1,n)
          return
     bary:print "La somme sur j de l-s à l+s de @1f vaut";dsum(@1
      ,j,l-s,l+s)
          return
```

Sortie (17 s)

```
  La somme sur les valeurs entières x de 1 à n de x est
    1/2*n^2 +1/2*n
  La somme sur les valeurs entières x de 1 à n de x^10 est
    1/11*n^11 +1/2*n^10 +5/6*n^9 -n^7 +n^5 -1/2*n^3 +5/66*n
  La somme sur les valeurs entières x de 1 à n de 1/x/(x+2)/(x+8) est
    1/13440* [n]* [n +8]^-1* [n +7]^-1* [n +6]^-1* [n +5]^-1* [n +4]^-1*
    [n +3]^-1* [n +2]^-1* [n +1]^-1* [919*n^7 +33084*n^6 +495054*n^5 +39
   78184*n^4 +18463431*n^3 +49218316*n^2 +69240596*n +39204016]
  La somme sur j de l-s à l+s de (2*j+1)*(j*(j+1)-s*(s+1)-l*(l+1)) vaut
    0
```

sdpoly(f, x)

V_fonction Sommation algébrique

f

polynôme en x

x

littéral

La fonction `sdpoly` renvoie le polynôme $F(x)$ qui vérifie l'équation (1) et tel que $F(x) = 0$ pour $x = 0$.

sdfrac(f, x)

V_fonction Sommation algébrique

f

expr

x

littéral

La fonction `sdfrac` renvoie l'expression $F(x)$ qui vérifie l'équation (1) et telle que $F(x) = 0$. S'il n'y a pas de solution arrêt après message.

sdrap(w, x)

V_fonction Sommation algébrique

w

expr

x

littéral

Pour sommer $f(x)$, donner comme argument w le rapport $f(x)/f(x-1)$. Si $F(x)/F(x-1)$ est rationnel, une solution $F(x)$ s'obtient par `sdrap`$(w, x) \times f(x)$.

La fonction `sdrap` permet la sommation d'expressions non rationnelles en x, comme par exemple $f(x) = x \times u^x$. On a en effet $f(x)/f(x-1) = xu/(x-1)$ et l'appel :

```
print sdrap(x*u/(x-1),x)
```

donne la solution :

$$F(x) = \frac{u^{x+1}(xu - x - 1)}{(u-1)^2}$$

mais il faut continuer le calcul de $F(b) - F(a-1)$ à la main, parce que le Basic 1000d n'accepte pas l'expression u^{x+1}.

Géométrie plane

Coordonnées

Les coordonnées $(X,\ Y)$ dans un repère orthonormé représentent un point A du plan euclidien. Les diverses fonctions et procédures utilisent, pour minimiser les temps de calcul, les coorpoly $(a_x,\ a_y,\ a_z)$ de A où a_x, a_y et a_z sont des polynômes tels que $X = a_x/a_z$ et $Y = a_y/a_z$. Par exemple les coorpoly $(ab,\ 1,\ b)$ ou $(-2ab,\ -2,\ -2b)$ représentent le même point $(a,\ 1/b)$. Les coorpoly $(x,\ y,\ z)$ sont dites réduites lorsque `gcd`$(x,\ y,\ z)$=1. Les coorpoly (qui sont des coordonnées projectives limitées à des polynômes) permettent de traiter les points à l'infini $(u,\ v,\ 0)$.

Les droites sont définies par la donnée de 2 points. Par exemple, la donnée de $(a,\ b,\ 1)$ et $(1,\ p,\ 0)$ définit la droite passant par $(a,\ b)$ et de pente p. Une droite peut être également définie par dualité par le point M de coorpoly $(x,\ y,\ z)$. Le point A $= (a_x,\ a_y,\ a_z)$ est un point de cette droite si et seulement si $xa_x + ya_y + za_z = 0$.

Notations

Les notations suivantes sont communes à tous les sous-programmes.

ax, ay, az, bx, by, bz, ...

expr

Les points A $= (a_x,\ a_y,\ a_z)$, B $= (b_x,\ b_y,\ b_z)$, ... désignent les entrées des diverses procédures. Les diverses grandeurs a_x, ... sont écrites sans indices `ax`, ... dans les descriptions de programmes.

x, y, z, x1, y1, y2, ...

nomi de type var (sortie)

Les points M $= (x,\ y,\ z)$, M$_1 = (x_1,\ y_1,\ z_1)$, ... sont les sorties des procédures. Les noms x, y, ... sont utilisés en accès, et doivent être déclarés de type var avant l'appel des procédures. Les coorpoly sorties sont réduites.

coorpoly X, Y, x, y, z

Procédure Coorpoly réduites du point M

X, Y

expr coordonnées de M (entrée)

La procédure `coorpoly` détermine les coorpoly réduites du point M.

Exemple

```
var x,y,z
print "On peut prendre pour coorpoly de (a,1+1/b)"
coorpoly a,1+1/b,x,y,z
print "x=";x
print "y=";y
print "z=";z
```

Sortie (630 ms)

```
On peut prendre pour coorpoly de (a,1+1/b)
x=  a*b
y=  b +1
z=  b
```

droite ax, ay, az, bx, by, bz, x, y, z

Procédure Point M dual de la droite AB

milieu ax, ay, az, bx, by, bz, x, y, z

Procédure Milieu M du segment AB

perpinf ax, ay, az, bx, by, bz, x, y, z

Procédure Point à l'infini M dans la direction perpendiculaire à AB.

Pour les procédures `droite`, `milieu` et `perpinf` il faut en entrée deux points distincts (A≠B). La sortie z de `perpinf` est 0.

Exemple

Médiatrice

```
var x0,y0,z0,x1,y1,z1
print "La médiatrice de A(a+7,b-2) B(c+h*4,b+d) est la
 droite passant par le milieu M de AB:"
milieu a+7,b-2,1,c+h*4,b+d,1,x0,y0,z0
print "(";x0/z0;"  , ";y0/z0;"  )"
print "et de pente"
perpinf a+7,b-2,1,c+h*4,b+d,1,x1,y1,z1
print y1/x1
print "C'est la droite d'équation"
droite x0,y0,z0,x1,y1,z1,x0,y0,z0
print x0*x+y0*y+z0;"=0"
```

Sortie (1065 ms)

```
La médiatrice de A(a+7,b-2) B(c+h*4,b+d) est la droite passant par le
 milieu M de AB:
```

```
( 1/2*a +1/2*c +2*h +7/2 ,  b +1/2*d -1 )
et de pente
  [d +2]^-1* [a -c -4*h +7]
C'est la droite d'équation
 -a^2 +2*a*x -14*a +2*b*d +4*b +c^2 +8*c*h -2*c*x +16*h^2 -8*h*x +d^2
 -2*d*y +14*x -4*y -53=0
```

interd ax, ay, az, bx, by, bz, cx, cy, cz, dx, dy, dz, x, y, z

Procédure Intersection M des droites AB et CD

Il faut A≠B et C≠D en entrée de la procédure `interd`. Dans le cas où les droites sont parallèles, M est le point à l'infini sur ces droites.

centrec ax, ay, az, bx, by, bz, cx, cy, cz, x, y, z

Procédure Centre M du cercle circonscrit au triangle ABC.

Il faut trois points distincts A, B, C en entrée de la procédure `centrec`.

dist2 ax, ay, az, bx, by, bz, n, d

Procédure Calcule le carré de la distance AB.

La procédure `dist2` renvoie les polynômes premiers entre eux n et d tels que $AB^2 = n/d$.

Exemple

Rayon d'un cercle

```
var x,y,z
print "Le carré du rayon du cercle passant par les poin
 ts (3,2), (a,5) et (7,a) est"
centrec 3,2,1,a,5,1,7,a,1,x,y,z
dist2 3,2,1,x,y,z,x,y
print formf(x/y)
```

Sortie (2840 ms)

```
Le carré du rayon du cercle passant par les points (3,2), (a,5) et (7,
 a) est
  1/2* [a -6]^-2* [a +1]^-2* [a^2 -12*a +37]* [a^2 -6*a +18]* [a^2 -4*
 a +20]
```

projorth ax, ay, az, bx, by, bz, cx, cy, cz, x, y, z

Procédure Projection orthogonale M de C sur la droite AB

Il faut A≠B en entrée de la procédure `projorth`.

aligne(ax, ay, az, bx, by, bz, cx, cy, cz)

V_fonction Teste l'alignement

La fonction `aligne` renvoie une expression nulle lorsque les trois points A, B et C sont alignés.

Exemple

Soit ABC un triangle. Pour tout point M du plan ABC, on désigne par P, Q et R les projections orthogonales de M sur les droites BC, CA et

AB. Déterminer le lieu des points M tel que les droites AP, BQ et CR soient concourantes.

Les coorpoly utilisées sont A = (a, b, 1), B = (c, d, 1), C = (e, f, 1), M = (u, v, 1), P = (A, B, C), Q = (D, E, F) et R = (G, H, I). Les coorpoly (da, db, dc), (dd, de, df) et (dg, dh, di) représentent les points duaux des droites AP, BQ et CR. La condition de concourance équivaut à la colinéarité de ces points, ce qui donne pour le lieu cherché une cubique. Nota : pour simplifier on a posé $a = b = c = 0$.

```
a=0
b=0
c=0
var A,B,C,D,E,F,G,H,I
var da,db,dc,dd,de,df,dg,dh,di
projorth c,d,1,e,f,1,u,v,1,A,B,C
projorth e,f,1,a,b,1,u,v,1,D,E,F
projorth a,b,1,c,d,1,u,v,1,G,H,I
droite a,b,1,A,B,C,da,db,dc
droite c,d,1,D,E,F,dd,de,df
droite e,f,1,G,H,I,dg,dh,di
W=aligne(da,db,dc,dd,de,df,dg,dh,di)
print "Le lieu est l'ensemble des points (u,v) tels que
 "
print W;" =0"
```

Sortie (2160 ms)

```
Le lieu est l'ensemble des points (u,v) tels que
  d^3*e^4*v +d^3*e^3*f*u -2*d^3*e^3*u*v +2*d^3*e^2*f^2*v -2*d^3*e^2*f*
 v^2 -d^2*e^5*u -2*d^2*e^4*f*v +d^2*e^4*u^2 -d^2*e^4*v^2 -d^2*e^3*f^2*
 u +2*d^2*e^3*f*u*v +2*d^2*e^3*u*v^2 -2*d^2*e^2*f^3*v +2*d^2*e^2*f*v^3
  +2*d*e^5*u*v +2*d*e^4*f*v^2 -2*d*e^4*u^2*v +2*d*e^3*f^2*u*v -4*d*e^3
 *f*u*v^2 +2*d*e^2*f^3*v^2 -2*d*e^2*f^2*v^3 =0
```

Tracé de courbes

fplot t1, t2, t3, x0, y0, sx, sy, fx, fy

Procédure Tracé de courbe

t1, t2, t3

réels, définissent le v_ensemble [t1, t2, t3]

x0, y0

réels, coordonnées absolues de l'origine

sx, sy

réels positifs, facteurs d'échelle

fx, fy

noms de V_fonctions

La procédure `fplot` permet de tracer la courbe plane donnée sous forme paramétrée par $x = \texttt{fx}(t)$, $y = \texttt{fy}(t)$. Le tracé est effectué à partir des points correspondants aux valeurs t du V_ensemble [t1, t2, t3]. Les points adjacents sont reliés par des segments de droite. La courbe est disposée de façon traditionnelle, avec y croissant vers le haut de l'écran. La valeur sx (resp sy) donne le nombre de pixels qui représentent la longueur 1 sur l'axe des x (resp y).

Exemple (monochrome)

Le programme trace en 15 s la néphroïde d'équation :

$$x = 3\sin t - \sin 3t$$
$$y = 3\cos t - \cos 3t.$$

La courbe est obtenue en faisant varier t de 0 à 2π avec un pas de $\pi/30$. L'origine est le point de coordonnées absolues écran (320, 232). L'unité vaut 50 pixels sur les axes Ox et Oy.

```
   fplot 0,2*pi,pi/30,320,232,50,50,fx,fy
   stop
fy:function(t)
   value=3*cos(t)-cos(3*t)
   return
fx:function(t)
   value=3*sin(t)-sin(3*t)
   return
```

axis x0, y0, x1, y1, x2, y2, dx, dy, x$, y$

Procédure Tracé d'axes

x0, y0

réels, coordonnées relatives de O

x1, y1, x2, y2

réels, coordonnées relatives d'un rectangle x1 < x2 et y1 > y2

dx, dy

réels dx > 0, dy > 0

x$, y$

exprchaînes

La procédure `axis` trace des axes orthogonaux se croisant en O et portant des marques numérotées espacées de dx ou dy pixels. Les textes x$ et y$ sont affichés aux extrémités des axes. Le tracé est limité au rectangle x1, y1, x2, y2.

Exemple (monochrome)

Le programme trace en 4 s des axes et la courbe $y = e^x$ pour x variant de -3.5 à 2.3 avec un pas de 0.2. Les fonctions entrées dans `fplot` sont ici les fonctions internes `float` et `exp`. L'origine est le point de coordonnées absolues écran (384, 370). L'unité vaut 120 pixels sur l'axe Ox et 40 pixels sur l'axe Oy.

```
axis 384,370,0,399,639,64,120,40,"x","exp(x)"
fplot -3.5,2.3,.2,384,370,120,40,float,exp
```

Fonctions eulériennes

La fonction Gamma $\Gamma(x)$ est calculée en utilisant le développement de Stirling à l'ordre 61. Ce développement a été obtenu par l'appel `stirling` 31. Le développement de Stirling est relié aux nombres de Bernoulli, qui peuvent être sortis par l'appel de la procédure `bernoulli`.

gamma(x)

V_fonction $\Gamma(x)$

x

réel

Tous les index et variables utilisés par la fonction `gamma` (ce nom est écrit en minuscules) sont locaux sauf la variable `gamma_psic`. Le premier appel initialise la table de données `gamma_psic` et est donc un peu plus long que les appels suivants. La précision est égale à la précision en cours sans toutefois pouvoir dépasser 166 chiffres décimaux. Lorsque l'argument x de `gamma` est entier, le calcul est également effectué via le développement de Stirling, ce qui permet de tester la précision de la méthode.

Exemple

La relation $\Gamma(x)\Gamma(1-x) = \pi/\sin\pi x$ est vérifiée pour $x = 1/3$.

```
precision 50
g1=gamma(1/3)
g2=gamma(2/3)
print g1
print g2
format -5
print g2*g1-pi/sin(pi/3)
```

Sortie (18 s)

```
0.26789385347077476336556929409746776441286893779573~ E+1
0.13541179394264004169452880281545137855193272660568~ E+1
0.3177~ E-52
```

stirling n

Procédure Calcul du développement de Stirling à l'ordre $2n-1$

n

entier $n > 0$

bernoulli

Procédure Sort les nombres de Bernoulli
Il faut d'abord appeler `stirling`, pour initialisations.

Exemple

On obtient, entre autres, le développement asymptotique :

$$\Gamma(x) = e^{-x}x^{x-1/2}\sqrt{2\pi}(1 + \frac{1}{12x} + \frac{1}{288x^2} - \frac{139}{51840x^3} - \frac{571}{2488320x^4} + \frac{163879}{209018880x^5} + \cdots)$$

```
stirling 3
bernoulli
```

Sortie (1450 ms)

```
Sortie du développement de Stirling de log Γ(x) à l'ordre   5
log Γ(x)= (x-1/2)*log(x)-x+log(2*pi)/2+ psi +O(1/x^7)
psi= (  1/12)/x+( -1/360)/x^3+(  1/1260)/x^5
timer=  0
Voici aussi le développement de Stirling de Γ(x)
Γ(x)=exp(-x)*x^(x-1/2)*sqr(2*pi)*{ (  1)+(  1/12)/x+(  1/288)/x^2+( -1
 39/51840)/x^3+( -571/2488320)/x^4+(  163879/209018880)/x^5  +O(1/x^6)
  }
timer=  1
Nombres de Bernoulli
B1=  1/6
B2=  1/30
B3=  1/42
```

gammap(a, x)

gammap(a, x, *)

Fonctions Gamma incomplètes

a, x

réels $a > 0$, $x \geq 0$

La forme `gammap`(a, x) calcule la fonction :

$$P(a,x) = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1}\, dt.$$

Pour a donné, $P(a,x)$ croît de 0 à 1 lorsque x varie de 0 à ∞. La forme `gammap`(a, x, *), où le troisième argument est arbitraire, renvoie $1 - P(a,x)$. Comme pour la fonction `gamma`, la précision est limitée à 166 chiffres.

Méthode

Pour $x < \min(8, a)$, la fonction est calculée par le développement en série de la fonction hypergéométrique F à partir de :

$$P(a,x) = e^{-x} x^a F(1, a+1, x)/\Gamma(a+1).$$

Sinon, le développement en fraction continue suivant est utilisé :

$$(1 - P(a,x))\Gamma(a) e^x x^{-a} = \cfrac{1}{x + \cfrac{1-a}{1 + \cfrac{1}{x + \cfrac{2-a}{1 + \cfrac{2}{x + \cdots}}}}}$$

Exemple (monochrome)

Le programme trace la courbe $y = P(4,x)$ en 35 s.

```
        axis 50,370,0,399,639,64,60,280,"x","P(4,x)"
        fplot 0,10,.3,50,370,60,280,float,gammapy
        stop
gammapy:function(x)
        value=gammap(4~,x)
        return
```

erf(x)

Fonction d'erreur

phistar(x)

Fonction de répartition normale

x

réel

La fonction `erf` calcule l'intégrale :

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

et la fonction `phistar` calcule :

$$\Phi^*(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-t^2/2}\, dt.$$

Méthode

Ces fonctions sont calculées à l'aide de la fonction `gammap`, par exemple `erf`(x) est donné par $P(1/2, x^2)$.

Exemple

Le premier appel est plus long, par suite des initialisations dans la fonction `gamma`.

```
print erf(1~);mtimer
clear timer
print erf(1~);mtimer
```

Sortie (3060 ms)

```
0.8427007929~  2220
0.8427007929~  835
```

Intégration numérique

romberg(a, b, f [, e])

V_fonction Intégrale $I = \int_a^b f(x)\,dx$

a, b, e

réels

Si $e > 0$ l'intégrale est calculée à e près. Si $e < 0$ l'intégrale I est calculée à $|e|I$ près. Par défaut $e = -2^{-\text{precision2}}$.

f

nom d'une fonction

Le nom `f` est tel que `f`(x) renvoie la valeur au point $x \in [a, b]$.
La fonction `romberg` calcule l'intégrale définie :

$$I = \int_a^b f(x)\,dx$$

par la méthode de Romberg. Tous les index et variables utilisés sont locaux. La fonction à intégrer `f` ne doit pas modifier les variables et index de `romberg`. Il est donc prudent de n'utiliser dans `f` que des éléments locaux.

Exemple

Calcul de la constante d'Euler par :

$$\gamma = \int_0^1 \frac{1 - e^{-t} - e^{-1/t}}{t} dt$$

```
print romberg(0,1,f1,10^-9)
```

```
        stop
     f1:function(tf)'fonction intégrée en exemple
        tf=float(tf)
        if tf<10^-5
          value=1-tf/2
          return
        else
          value=(1-exp(-tf)-exp(-1/tf))/tf
          return
        endif
```

Sortie (24 s)

```
0.5772156649~
```

Exemple

Calcul de

$$\mathrm{li}(x) = \int_2^x \frac{1}{\log x} dx.$$

La fonction li(x) est une estimation du nombre de nombres premiers inférieurs à x. Le calcul est effectué en donnant l'argument `1/log` dans `romberg`, ce qui est autorisé.

```
format 1
li=0
b=2
for i=1,17
  a=b
  read b
  data 10,100,1000,10000,50000,100000,500000,1000000,20
   00000,5000000
  data 10000000,20000000,90000000,100000000,200000000,5
   00000000,1000000000
  li=li+romberg(a,b,1/log,.5)
  print "li(";b;")=";li
next i
```

Sortie (18 s)

```
li(  10)=  5.~
li(  100)=  29.~
li(  1000)=  177.~
li(  10000)=  1245.~
...
li(  500000000)=  26356831.~
li(  1000000000)=  50849234.~
```

Racines réelles d'un polynôme

Les sous-programmes suivants déterminent toutes les racines réelles d'un polynôme f unilittéral en x (par exemple), avec leurs multiplicités. On obtient des valeurs exactes pour les racines rationnelles, et des valeurs flottantes, approchées à mieux que $2^{-\text{precision2}}$ en valeur relative, pour les racines irrationnelles.

zerop f, q

zero f, T, M, p, q

Procédures Racines réelles

f

poly unilittéral

q

entier*32.
Si $q \neq 0$, f est d'abord complètement factorisé dans $\mathbf{Q}[x]$. Il est en général avantageux, pour le temps de calcul, d'utiliser $q \neq 0$. On conseille donc de n'utiliser $q = 0$ que si f est connu comme irréductible. De même si une factorisation de f est connue, il vaut mieux entrer cette forme factorisée.

p

nomi de type var ou index
p contient en sortie le nombre de racines distinctes.

T, M

nomi de type var qui doivent être déclarés par :

```
var T(n),M(n)
```

où `n` est au moins égal au nombre de racines distinctes (la valeur `n` = `degf`(f, x) convient toujours).
La procédure `zero` renvoie les valeurs des p racines dans les variables T(1), ..., T(p) et les multiplicités correspondantes dans M(1), ..., M(p). La procédure `zerop` se contente d'écrire les racines. Noter que si $q \neq 0$, les racines rationnelles sont sorties sous forme exacte, par contre si $q = 0$, des racines rationnelles peuvent sortir sous forme flottante.

Méthode

Algorithme de Collins et Akritas
Voir la description et la comparaison à d'autres méthodes dans *Computer Algebra* (1982) p83.

Exemple

Détermination, avec 20 chiffres exacts, des zéros d'un polynôme de degré 6. Il y a une racine double rationnelle.

```
w=37538*x^6 -236736*x^5 +482848*x^4 -357108*x^3 +97519*
 x^2 -11046*x +441
precision 20
zerop w,1
```

Sortie (52 s)

```
Zéros réels de f=  37538*x^6 -236736*x^5 +482848*x^4 -357108*x^3 +9751
 9*x^2 -11046*x +441
Nombre de zéros distincts=  5
comptés avec leur (multiplicité)=  6
(1)    0.28989663259659067020~ E+1
(1)    0.22368128791039502966~ E+1
(1)    0.76318712089604970342~
(2)    0.15328467153284671533~  exact=  21/137
(1)    0.10103367403409329797~
```

Performances

Nous donnons ici le temps de calcul (en s) des k racines du polynôme irréductible de degré k, `ppwr`(x, k)+1, avec p chiffres exacts pour divers k et p.

k	$p = 10$	20	30	40	50	100	200	300	400
5	31	63	111	177	269	1183	6783	20427	37796
10	113	222	398	659	1042	3252	18766		
15	305	609	1155	2027	3356	9151			
20	721	1445	2837	5141	8735				
25	1709	3269	6386	11656	19956				

zerob A, B, f

f

polynôme irréductible unilittéral en x, de degré ≥ 2

Le nom du littéral de f doit ici vraiment être donné par x, à la différence de `zero` et `zerop`.

A, B

nomi de type var (variables de sortie)

La procédure `zerob`, qui est utilisée par `zero` et `zerop`, présente un intérêt en soi. Elle renvoie $p =$ `deg`(B, x) intervalles $]a_i, b_i[$ isolant les p racines réelles positives de f. Les intervalles sont renvoyés codés dans les deux polynômes A et

B :

$$A = \sum_{i=1}^{p} a_i x^i$$

$$B = \sum_{i=1}^{p} b_i x^i$$

La procédure est exacte, et ne dépend pas de la précision. En fait la procédure accepte que f ne soit pas irréductible, à condition que f n'ait aucune racine multiple. Dans ce cas il se peut alors qu'en plus des segments distincts $]a_i, b_i[$, apparaissent des points codés par $b_k = a_k$, qui peuvent être inclus dans d'autres segments, et correspondent à des racines exactes. Ces points sont renvoyés, en même temps que les segments isolant les racines dans A et B. Si on veut isoler toutes les racines réelles de f, positives et négatives, il faudra appeler la procédure 2 fois avec $f(x)$ puis `subs`(f,$x = -x$) comme entrée.

Optimisation

But

Trouver le point $(x_1,\ x_2,\ \ldots,\ x_n)$ où la fonction réelle de n variables réelles $f(x_1,\ x_2,\ \ldots,\ x_n)$ atteint son minimum.

Méthode

Méthode du simplexe, qui ne nécessite que le calcul de la fonction, pas de ses dérivées. Voir J A Nelder & R Mead (1965). Pour le plaisir du lecteur, voici la description de l'algorithme, avec les notations utilisées dans le programme. Par exemple n_{par} correspond au nom `npar` dans le code.

Simplexe initial

A partir des n_{par} valeurs initiales $par(i)$ $(i \in [1, n_{\text{par}}])$ et des accroissements initiaux $accr(i)$ $(i \in [1, n_{\text{par}}])$ on construit un simplexe (ensemble de $n_{\text{par}} + 1$ points). Les coordonnées du point j $(j \in [0, n_{\text{par}}])$ sont $pnt(i, j) = par(i) + accr(i)\delta_{i,j}$ $(i \in [1, n_{\text{par}}])$.

Etape 2

On calcule la valeur de la fonction pour les points du simplexe (la valeur pour le point j est mise dans $pnt(0, j)$).

Etape 3

On détermine les points $i_{\min}$ et $i_{\max}$ où la fonction prend sa valeur minimum $w_{\min}$ et maximum $w_{\max}$ sur le simplexe. On calcule également le

centre de gravité G des n_{par} meilleurs points du simplexe (ses coordonnées sont $grav(i)$).

Convergé?

Si $w_{\max} - w_{\min} < eps$ on estime que le minimum est atteint et on arrête le calcul.

Symétrique

On calcule la valeur w_{sym} de la fonction au point symétrique i_{sym} de $i_{\max}$ par rapport à G.

Si $w_{\min} < w_{\text{sym}} < w_{\max}$ on remplace le point $i_{\max}$ par i_{sym} pour former un nouveau simplexe, et on continue à l'étape 3.

Plus loin

Si $w_{\text{sym}} < w_{\min}$ on estime que la direction était bonne, et on calcule la valeur de la fonction w_{pln} au point i_{pln} symétrique de G par rapport à i_{sym}. Puis on remplace le point $i_{\max}$ par le meilleur des points i_{sym} ou i_{pln}, et on continue à l'étape 3.

Milieu

Si $w_{\max} < w_{\text{sym}}$ on estime que la direction n'était pas bonne, et on calcule la valeur de la fonction w_{mil} au point i_{mil} milieu de $i_{\max}$ et G. Si $w_{\text{mil}} < w_{\max}$ on remplace le point $i_{\max}$ par i_{mil} et on continue à l'étape 3.

Contraction

On interprète le fait que i_{sym} et i_{mil} soient plus mauvais que $i_{\max}$ en pensant que $i_{\min}$ est proche du minimum. On remplace alors le simplexe initial par le simplexe homothétique de rapport 1/2 par rapport au point $i_{\min}$, et on continue à l'étape 2.

En général la méthode converge, mais peut renvoyer un minimum local. Il est donc conseillé de l'appeler plusieurs fois avec des points et accroissements initiaux différents.

simplex(npar, eps, par(1), accr(1), f [, *])

Fonction Recherche du minimum de la fonction *f*

npar

entier $n_{\text{par}} > 0$ nombre de paramètres

eps

petit réel $eps > 0$

Pour la signification de *eps* voir l'étape Convergé? de l'algorithme. Si on donne une valeur trop petite, la fonction `simplex` peut boucler.

par(1)

Elément d'une variable indicée de type var

$par(1), par(2), \ldots, par(n_{\text{par}})$ doivent contenir en entrée les estimations initiales. En sortie, ces variables contiendront les coordonnées du minimum. Noter que

cet argument est utilisé en accès. Au lieu de *par*(1) on peut utiliser *par*(25) ou *par*(3,4,2) par exemple.

accr(1)

Elément d'une variable indicée de type var
accr(1), ..., *accr*(n_{par}) doivent contenir en entrée les accroissements initiaux.

f

fonction à minimiser
`simplex` va appeler cette fonction, en lui transmettant les coordonnées d'un point. La fonction `f` doit renvoyer la valeur en ce point. On utilisera donc un programme du type suivant :

```
f:function(access x(npar-1))
```

Le nombre de variables `npar` $= n_{\text{par}}$ est initialisé par le sous-programme `simplex` et x(0), x(1), ..., x($n_{\text{par}} - 1$) donnent les coordonnées du point que la fonction `f` ne doit pas modifier.

[,*]

Si un sixième argument est donné, `simplex` écrit ses étapes, le numéro d'itération, la valeur de la fonction et les minimums trouvés.

Sorties

La fonction `simplex` renvoie la valeur du minimum trouvé. Les coordonnées du minimum sont dans *par*(1), ..., *par*(n_{par}). Le nombre de fois que la fonction a été calculée est `simplex_iter`.

Performances

La minimisation de $\sqrt{\sum_{i=1}^{n}(x_i - 1)^2}$ est effectuée en partant de $par(i) = 0$, avec des accroissements $accr(i) = 1/10$ et pour $eps = 10^{-10}$. Voici pour divers $n = n_{\text{par}}$, le minimum atteint m, le nombre d'itérations i et le temps du calcul t.

n	m	i	t(s)
1	.7E-10	166	24
2	.1E-09	299	92
3	.1E-09	485	252
5	.1E-09	1002	1078
10	.5E-09	7780	29114

Exemple

Le programme qui suit correspondant à $n = 2$, c'est à dire à la minimisation de $\sqrt{(x_1 - 1)^2 + (x_2 - 1)^2}$. Le temps de sortie diffère de 45 s de celui dans la table ci-dessus par suite de l'affichage des étapes de l'optimisation. Pour chaque étape on obtient le type de point calculé, le numéro de l'étape et la valeur de la fonction, et pour chaque meilleur minimum, la valeur des paramètres.

```
var par(10),accr(10)
```

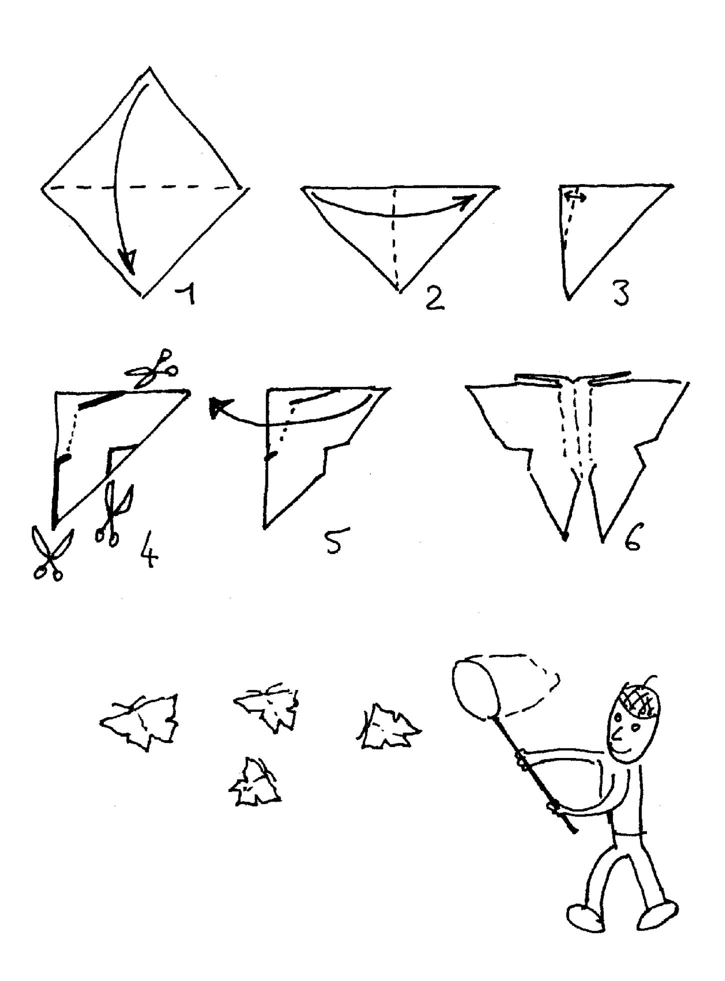
```
          accr(1)=.1
          accr(2)=.1
          print simplex(2,10~^-10,par(1),accr(1),ftest,+)
          stop
    ftest:function(access par(npar-1))
          local index i
          value=sqr(sum(i=1,npar of (par(i-1)-1)^2))
          return
```

Sortie (137 s)

```
  init                 1           0.1414213562~ E+1
  init                 2           0.1345362405~ E+1
  init                 3           0.1345362405~ E+1
  meilleur          0.1345362405~ E+1  1.0000000000~ E-1  0
  sym                  4           0.1272792206~ E+1
  +loin                5           0.1131370850~ E+1
  meilleur          0.1131370850~ E+1  0.2000000000~  0.2000000000~
  ...
  sym                281           0.1492325435~ E-9
  +loin              282           0.2110829576~ E-8
  meilleur          0.1492325435~ E-9  0.1000000000~ E+1  0.1000000000~
   E+1
  ...
  fin                299           0.1492325435~ E-9
    0.1492325435~ E-9
```

17

Exemples d'application



Nous étudions d'abord à l'aide du Basic 1000d un système d'équations, qui correspond à un problème d'électronique, à savoir régler les phases d'oscillateurs pour éliminer les harmoniques 3, 5 et 7. Dans cette étude, la bibliothèque MATH est utilisée. Ensuite nous vous proposons quelques exercices de programmation.

Résolution d'un système d'équations

Mise en forme du problème

Soit à résoudre le système d'équations :

$$\begin{cases} \sin 3x + \sin 3y + \sin 3z = 0 \\ \sin 5x + \sin 5y + \sin 5z = 0 \\ \sin 7x + \sin 7y + \sin 7z = 0 \end{cases}$$

Nous allons d'abord changer d'inconnues, au lieu de x, y, z, nous utiliserons $x_1 = \sin x$, $x_2 = \sin y$ et $x_3 = \sin z$. Il faut d'abord exprimer $\sin nx$ ($n = 3$, 5 ou 7) en fonction de $x_1 = \sin x$. Le programme suivant effectue ce calcul. Les littéraux `c` et `s` représentent $\cos x$ et $\sin x$. La variable `sn`, qui représente $\sin nx$, s'obtient comme partie imaginaire du nombre complexe `en` $= \exp(inx) =$ `(c+i*s)^n`. Elle est écrite en fonction de `s` par la substitution $\mathtt{c}^2 \to 1 - \mathtt{s}^2$.

```
complex i
for n=3 to 7 step 2
  en=(c+i*s)^n
  sn=Im(en)
  sn=subsrr(sn,c^2=1-s^2)
  print "sin(";justl$(n);"x)=";change$(sn,"s","sin(x)")
next n
```

Résultat (835 ms)

$$\sin 3x = -4\sin^3 x + 3\sin x$$

$$\sin 5x = 16\sin^5 x - 20\sin^3 x + 5\sin x$$

$$\sin 7x = -64\sin^7 x + 112\sin^5 x - 56\sin^3 x + 7\sin x$$

Le programme suivant écrit (en 1 s) le système à résoudre suivant les inconnues x_i ($i = 1$, 2, 3).

```
lit x(3)
w3=sum(i=1,3 of -4*x(i)^3 +3*x(i))
w5=sum(i=1,3 of 16*x(i)^5 -20*x(i)^3 +5*x(i))
w7=sum(i=1,3 of -64*x(i)^7 +112*x(i)^5 -56*x(i)^3 +7*x(
```

```
 i))
print w3;"=0"æ
print w5;"=0"æ
print w7;"=0"æ
```

Une méthode trop brutale

Essayons la résolution de ce système de trois inconnues à trois équations, par élimination. Le programme suivant, élimine d'abord x_3, donnant deux équations en x_1 et x_2 de degrés 15 et 21 respectivement.

La factorisation des polynômes obtenus montre qu'ils sont divisibles par $x_1x_2(x_1 + x_2)$. Les solutions annulant ce facteur étant aisées à déterminer, on a divisé par ce facteur les deux équations, réduisant les degrés à 12 et 18. L'élimination de x_2 entre ces deux équations reste encore dans les possibilités du 1040ST. On obtient une équation en x_1 de degré 184. Noter qu'il a fallu redéfinir `s_var` (le dernier appel `elim` nécessite $(12 + 18)^2$ variables internes).

C'est pour poursuivre la résolution que la mémoire de l'ordinateur se trouve être insuffisante. En effet, après avoir récrit l'équation en terme de x_1^2, et avoir divisé par le facteur $(4x_1^2 - 3)^2$ (Ce facteur ne donne pas de solution du système initial), il reste une équation de degré 90. La suite de la résolution devrait être la factorisation de l'équation, ce qui provoque une erreur Mémoire.

On peut alors envisager une résolution numérique, mais la fonction `zero` de la bibliothèque MATH provoque également une erreur Mémoire.

```
s_var 1500
lit x(3)
w3=sum(i=1,3 of -4*x(i)^3 +3*x(i))
w5=sum(i=1,3 of 16*x(i)^5 -20*x(i)^3 +5*x(i))
w7=sum(i=1,3 of -64*x(i)^7 +112*x(i)^5 -56*x(i)^3 +7*x(
 i))
print "L'élimination de x(3) donne les deux équations"
w7=red(elim(w7,w3,x(3)))
w7=dive(w7,x(1)*x(2)*(x(1)+x(2)))
print "équation de degré=";deg(w7),left$(w7,200);" ..."
 ;"   timer=";timer
w5=red(elim(w5,w3,x(3)))
w5=dive(w5,x(1)*x(2)*(x(1)+x(2)))
print "équation de degré=";deg(w5),left$(w5,200);" ..."
 ;"   timer=";timer
print "Elimination de x(2)"
w7=elim(w7,w5,x(2))
w7=red(w7)
print "équation de degré=";deg(w7),left$(w7,200);" ..."
 ;"   timer=";timer
```

```
Sortie (4h1/2)
MODIFICATION DE STRUCTURE
```

```
CLEAR EFFECTUE
L'élimination de x(3) donne les deux équations
équation de degré=  18 ...
équation de degré=  12 ...
Elimination de x(2)
équation de degré=  184 ...
```

Utilisation des symétries

Le système d'équations est invariant dans les permutations de x_1, x_2 et x_3. On peut mettre à profit cette symétrie pour résoudre le système. Introduisons les fonctions symétriques :

$$s_1 = x_3 + x_2 + x_1$$
$$s_2 = x_3x_2 + x_3x_1 + x_2x_1$$
$$s_3 = x_3x_2x_1$$

Nous allons récrire le système à résoudre en fonction de ces fonctions symétriques. Les solutions x_i s'obtiendront ensuite comme racines de l'équation de degré 3 en u :

$$u^3 - s_1u^2 + s_2u - s_3 = 0.$$

Le programme suivant utilise la fonction `symf` de la bibliothèque MATH pour récrire le système d'équations en fonction de s_i. En entrée de cette fonction, on donne les expressions symétriques à l'aide de la variable `sv`. Le littéral d'homogénéisation `Z` est ensuite mis égal à 1.

```
lit x(3)
w3=sum(i=1,3 of -4*x(i)^3 +3*x(i))
w5=sum(i=1,3 of 16*x(i)^5 -20*x(i)^3 +5*x(i))
w7=sum(i=1,3 of -64*x(i)^7 +112*x(i)^5 -56*x(i)^3 +7*x(
 i))
var sv(3)
lit s(3)
sv(1)=  x(3) +x(2) +x(1)
sv(2)=  x(3)*x(2) +x(3)*x(1) +x(2)*x(1)
sv(3)=  x(3)*x(2)*x(1)
e3=subs(symf(w3,3,x,s,sv,Z),Z=1)
e5=subs(symf(w5,3,x,s,sv,Z),Z=1)
e7=subs(symf(w7,3,x,s,sv,Z),Z=1)
print e3;"=0"æ
print e5;"=0"æ
print e7;"=0"æ
```

Sortie (7 s)

```
-12*s(3) +12*s(2)*s(1) -4*s(1)^3 +3*s(1)=0

-80*s(3)*s(2) +80*s(3)*s(1)^2 -60*s(3) +80*s(2)^2*s(1) -80*s(2)*s(1)^
3 +60*s(2)*s(1) +16*s(1)^5 -20*s(1)^3 +5*s(1)=0
```

```
-448*s(3)^2*s(1) -448*s(3)*s(2)^2 +1344*s(3)*s(2)*s(1)^2 -560*s(3)*s(
2) -448*s(3)*s(1)^4 +560*s(3)*s(1)^2 -168*s(3) +448*s(2)^3*s(1) -896*
s(2)^2*s(1)^3 +560*s(2)^2*s(1) +448*s(2)*s(1)^5 -560*s(2)*s(1)^3 +168
*s(2)*s(1) -64*s(1)^7 +112*s(1)^5 -56*s(1)^3 +7*s(1)=0
```

La résolution du nouveau système commence par l'élimination de s_3. On obtient alors un système de deux équations en s_1 et s_2. Le programme suivant, écrit ces deux équations sous forme factorisée.

```
lit s(3)
e3= -12*s(3) +12*s(2)*s(1) -4*s(1)^3 +3*s(1)
e5= -80*s(3)*s(2) +80*s(3)*s(1)^2 -60*s(3) +80*s(2)^2*s
 (1) -80*s(2)*s(1)^3 +60*s(2)*s(1) +16*s(1)^5 -20*s(1)^
 3 +5*s(1)
e7= -448*s(3)^2*s(1) -448*s(3)*s(2)^2 +1344*s(3)*s(2)*s
 (1)^2 -560*s(3)*s(2) -448*s(3)*s(1)^4 +560*s(3)*s(1)^2
  -168*s(3) +448*s(2)^3*s(1) -896*s(2)^2*s(1)^3 +560*s(
 2)^2*s(1) +448*s(2)*s(1)^5 -560*s(2)*s(1)^3 +168*s(2)*
 s(1) -64*s(1)^7 +112*s(1)^5 -56*s(1)^3 +7*s(1)
e5=red(elim(e5,e3,s(3)))
e7=red(elim(e7,e3,s(3)))
print formf(e5);"=0"æ
print formf(e7);"=0"æ
```

Sortie (3 s)

```
[s(1)]* [40*s(2)*s(1)^2 -30*s(2) -16*s(1)^4 +30*s(1)^2 -15]=0

[s(1)]* [1344*s(2)^2*s(1)^2 -1008*s(2)^2 -1344*s(2)*s(1)^4 +2688*s(
2)*s(1)^2 -1260*s(2) +320*s(1)^6 -1008*s(1)^4 +1008*s(1)^2 -315]=0
```

Cas $s_1 = 0$

Il apparaît ainsi d'abord la solution suivante de ce système :

$$s_1 = 0$$
$$s_2 \text{ arbitraire}$$

La valeur correspondante de s_3 s'obtient en reportant $s_1 = 0$ dans l'équation `e3`=0, ce qui donne $s_3 = 0$. L'équation correspondante en u est :

$$u^3 + s_2 u = 0.$$

Les solutions en x_i sont (à une permutation près) :

$$x_1 = 0 \quad \text{ce qui donne } x = 0 \pmod{\pi}$$
$$x_2 = -x_3$$

Cas $s_1 \neq 0$

On poursuit la résolution, en supprimant le facteur $s_1 \neq 0$ des équations et en éliminant s_2. On obtient une équation en s_1, que l'on factorise. Cela est réalisé par le programme suivant :

```
lit s(3)
e5=40*s(2)*s(1)^2 -30*s(2) -16*s(1)^4 +30*s(1)^2 -15
e7= 1344*s(2)^2*s(1)^2 -1008*s(2)^2 -1344*s(2)*s(1)^4 +
 2688*s(2)*s(1)^2 -1260*s(2) +320*s(1)^6 -1008*s(1)^4 +
 1008*s(1)^2 -315
e7=elim(e7,e5,s(2))
print formf(e7);"=0"
```

Résultat (6 s)

$$-4(4s_1^2 - 3)(256s_1^8 - 2880s_1^6 + 10080s_1^4 - 12600s_1^2 + 4725) = 0$$

L'équation obtenue est de degré 10 en s_1, mais c'est en fait seulement une équation de degré 5 en s_1^2. La factorisation obtenue donne les deux cas suivants :

Cas $s_1^2 = 3/4$

En reportant cette valeur dans `e5`, par la commande :

```
print subsr(e5,s(1)^2=3/4)
```

on obtient la valeur $-3/2$ pour `e5`, c'est à dire qu'il n'y a pas de solution correspondant à ce cas.

Cas $256s_1^8 - 2880s_1^6 + 10080s_1^4 - 12600s_1^2 + 4725 = 0$

Dans ce cas, comme l'équation ne se factorise pas de façon rationnelle, nous poursuivons l'étude de façon numérique. La procédure `zero` de la bibliothèque MATH nous permet de déterminer les racines de l'équation. Ensuite, en reportant ces valeurs dans `e5`, puis `e3`, on obtient les valeurs numériques des solutions en s_1, s_2 et s_3. Noter que les calculs nécessitent la conversion des valeurs flottantes renvoyées par `zero` en valeurs exactes (on a utilisé `appr`).

La résolution se poursuit, en appelant de nouveau `zero`, avec pour entrée l'équation de degré 3 en u, ce qui donne les solutions en x_1, x_2 et x_3 à une permutation près.

Nous donnons seulement les solutions correspondant à $s_1 > 0$. Les solutions $s_1 < 0$ s'obtiennent en changeant les signes des x_i. A chaque valeur de s_1 il correspond des valeurs réelles des inconnues x, y, z du système de départ, dont nous donnons un seul exemple. Si x, y, z est une solution correspondant aux x_i, on obtient toutes les solutions en remplaçant x par $k\pi - x$ ou par $x + k\pi$ (k entier), et des remplacements analogues pour y et z.

```
lit s(3)
e3= -12*s(3) +12*s(2)*s(1) -4*s(1)^3 +3*s(1)
e5=40*s(2)*s(1)^2 -30*s(2) -16*s(1)^4 +30*s(1)^2 -15
e7=256*s(1)^8 -2880*s(1)^6 +10080*s(1)^4 -12600*s(1)^2
 +4725
```

```
var T7(8),TU(3),M(8),nb
zero e7,T7,M,nb,0
notilde
format 10
for i=1,4
  s1=appr(T7(i))
  ep5=subs(e5,s(1)=s1)
  ep3=subs(e3,s(1)=s1)
  s2=sroot(ep5,s(2))
  ep3=subs(ep3,s(2)=s2)
  s3=sroot(ep3,s(3))
  print "cas";i;"   s1=";float(s1);"   s2=";float(s2);"
      s3=";float(s3)
  zero u^3-s1*u^2+s2*u-s3,TU,M,nb,0
  print "   x(1)=";TU(1);
  print "   x(2)=";TU(2);
  print "   x(3)=";TU(3)
  A=asin(TU(1))
  B=asin(TU(2))
  C=asin(TU(3))
  print "(par exemple x=";A;"   y=";B;"   z=";C;")"
next i
```

Sortie (92 s)

```
cas  1   s1=  2.429215139   s2=  1.917714494   s3=  0.487508896
   x(1)=  0.979327550   x(2)=  0.891508709   x(3)=  0.558378879
(par exemple x=  1.367109951   y=  1.100664790   z=  0.592430380)

cas  2   s1=  1.797580965   s2=  0.857631992   s3=  0.054885330
   x(1)=  0.978176699   x(2)=  0.743986452   x(3)=  0.075417814
(par exemple x=  1.361496817   y=  0.839016712   z=  0.075489492)

cas  3   s1=  1.204209233   s2=  0.183610207   s3= -0.059925167
   x(1)=  0.941634011   x(2)=  0.415674653   x(3)= -0.153099431
(par exemple x=  1.227451720   y=  0.428684450   z= -0.153703923)

cas  4   s1=  0.817004656   s2= -0.637528121   s3= -0.498394891
   x(1)=  0.922371535   x(2)=  0.684280777   x(3)= -0.789647656
(par exemple x=  1.174175214   y=  0.753616949   z= -0.910234524)
```

Exercices de programmation

Exercice Mille

Calculer le produit des 1000 premiers nombres premiers.

Exercice Hilbert

Calculer le déterminant de la matrice 10×10 d'éléments $A_{ij} = 1/(i+j)$.

Exercice 11...11

Chercher des nombres premiers dont les chiffres (en base 10) sont tous des 1 (comme par exemple 11 et 11111111111111111111111).

Exercice Goldbach

En 1742, Goldbach proposa la conjecture suivante.
Tout nombre pair > 3 est la somme de deux nombres premiers. Par exemple :

$$4 = 2 + 2, \qquad 6 = 3 + 3, \qquad 8 = 3 + 5.$$

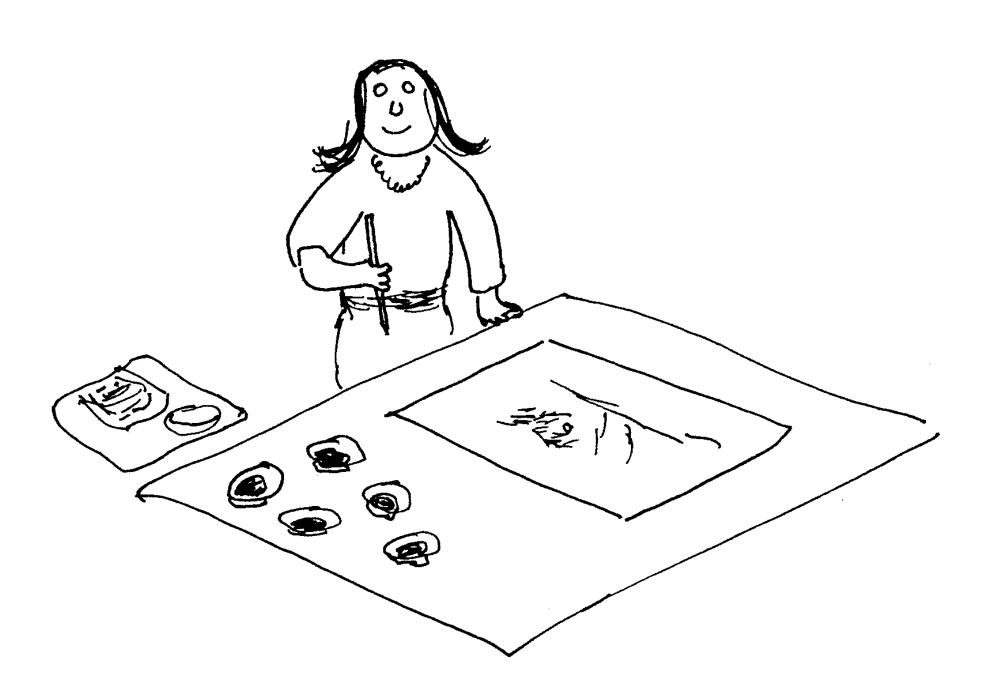
Cette conjecture n'est de nos jours ni démontrée ni infirmée. Ecrire un programme vérifiant la conjecture (ou essayant de trouver un contre exemple) par calcul numérique.

Exercice Parfait

Ecrire un programme recherchant les nombres parfaits pairs (voir l'exercice Grandissime pour des indications supplémentaires). S'il y a un nombre fini ou non de nombres parfaits, et s'il existe des nombres parfaits impairs sont encore des problèmes non résolus.

18

Graphismes



Coordonnées et Couleurs

Le système de coordonnées graphiques dépend de la résolution. Les points de l'écran (x, y) ont des coordonnées x variant de 0 (gauche) à $x_{\max} - 1$ (droite) et y variant de 0 (haut) à $y_{\max} - 1$ (bas). Le nombre de couleurs c dépend également de la résolution.

	`resolution`	c	$x_{\max}$	$y_{\max}$
Basse résolution	0	16	320	200
Moyenne résolution	1	4	640	200
Haute résolution	2 ou 3	2	640	400

ORIGINX [a]

ORIGINY [b]

Variables d'état Origine graphique

ORIGIN a, b

Commande Fixe l'origine graphique

a, b

entier*16

Ces commandes fixent l'origine du système de coordonnées. Lorsque l'origine est en (a, b), la donnée des coordonnées (x, y) d'un point dans les commandes graphiques correspond au point de coordonnées absolues ($a + x$, $b + y$). Toutes les instructions graphiques (exceptées `origin`, `originx` et `originy`), et les appels systèmes `vdir` et `vdirf` (mais pas `vdi` ou `vdif`), utilisent ces coordonnées relatives. Les coordonnées relatives peuvent être des réels non entiers, qui sont alors arrondis aux plus proches entiers.

Exemple

Le programme suivant utilise les variables d'état `originx` et `originy` d'abord en tant que commandes pour fixer l'origine au milieu de l'écran (en monochrome), et ensuite en tant que fonctions pour afficher en mode graphique les coordonnées absolues de l'origine. Ensuite un disque de rayon 5 est tracé autour de l'origine. Nota : Les sorties graphiques des exemples ne sont pas reproduites dans ce manuel.

```
originx 320
originy 200
text -35,25,"(" &justl$(originx) &"," &justl$(originy)
```

```
 &")"
pcircle 0,0,5
```

Sortie (555 ms)
On aurait pu utiliser la commande `origin 320, 200` au lieu des deux premières lignes du programme ci-dessus.

CLIP x, y, xp, yp [, f]

Commande Restriction de l'affichage

x, y, xp, yp

réels, coordonnées relatives de 2 sommets opposés d'un rectangle

f

entier (défaut $f = -1$)

Si $f \neq 0$, la commande `clip` limite les sorties graphiques au rectangle. Si `clip` est appelé avec $f = 0$, ou après `clear`, on revient à l'affichage normal.

Exemple

Une fenêtre est créée (`aes(100)`), ouverte (`aes(101)`) et les coordonnées du rectangle intérieur sont déterminées (`aes(104)`). L'origine est placée au coin haut gauche, et `clip` restreint l'affichage à l'intérieur de la fenêtre. La commande `on menu message` permet la surveillance du cliquage de la case de fermeture de la fenêtre. On demande par `circle` le tracé de cercles aléatoires, mais seul l'arc dans la fenêtre est vraiment tracé, par suite de l'appel de `clip`. De temps à autre la fenêtre est vidée par `pbox`, le type de remplissage `f_type` étant nul. Enfin, après cliquage de la case de fermeture, `aes(102)` ferme la fenêtre et `aes(103)` la supprime.

```
hidec
aes 100,2,50,50,150,100
f=peekw(gintout)
aes 101,f,50,50,150,100
aes 104,f,4
x=peekw(gintout+2)
y=peekw(gintout+4)
dx=peekw(gintout+6)
dy=peekw(gintout+8)
origin x,y
clip 0,0,dx-1,dy-1,1
on menu message fen
do
  on menu
  circle random(150),random(100),30+random(50)
  if random(10)=0
    f_type 0
    pbox 0,0,dx,dy
```

```
          endif
        loop
    fen:ift menu(1)<>22 return
        aes 102,f
        aes 103,f
        cls
        stop
```

COLOR(n)

VCOLOR(i)

Variables d'état Couleur de numéro n ou d'index i

n, i

entiers*16 $n \in [0, 15]$, $i \in [0, 15]$

En tant que commande, la syntaxe est (signe égal permis) :

```
color(n)=rvb
vcolor(i)=rvb
```

ou :

```
color(n) rvb
vcolor(i) rvb
```

rvb

entier*16

Une couleur est codée par ses proportions de rouge, vert et bleu r, v et b (de 0 à 7) à l'aide de l'entier $rvb = 2^8r + 2^4v + b$. Voici quelques exemples (noter que la base 16 est commode) :

rvb	couleur	rvb	couleur	rvb	couleur
$000	noir	$777	blanc	$333	gris
$700	rouge	$077	cyan	$733	rose
$070	vert	$707	magenta	$373	vert clair
$007	bleu	$770	jaune	$407	violet

Le numéro n de la couleur va de 0 à 1 (monochrome), 3 (résolution moyenne) ou 15 (basse résolution). La fonction `color`(n) renvoie le code rvb de de la couleur numéro n. Elle est équivalente à :

```
peekw($FF8240+2*n) and $777
```

En réalité `color` utilise la fonction `xbios(7)`. La commande :

```
color(n)=rvb
```

modifie la couleur de numéro n. En haute résolution, seulement la couleur numéro 0 et la parité de rvb sont actifs. On ne dispose donc que des possibilités :

```
color(0)=0
```

```
color(0)=1
```

Exemple (monochrome)

Inverse le noir et le blanc, revient en normal après appui sur une touche.

```
locate 10,10
print "Taper une touche"
color(0)=0
i=keyget
color(0)=1
```

Les instructions graphiques du Basic 1000d, ainsi que le VDI, utilisent pour fixer une couleur non pas son numéro n, mais son index i qui prend également les valeurs de 0 à 15. La variable d'état `vcolor`(i), analogue à `color`(n), utilise l'index i et non le numéro n de la couleur. Voici la table des correspondances entre numéro et index :

	basse résolution															
numéro	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
index	0	2	3	6	4	7	5	8	9	10	11	14	12	15	13	1

	résolution moyenne				haute résolution	
numéro	0	1	2	3	0	1
index	0	2	3	1	0	1

Exemple (basse résolution)

Le programme suivant, en basse résolution, montre une façon d'obtenir la table ci-dessus. Pour chaque valeur i de l'index, on remplit un rectangle et on lit le numéro n de la couleur d'un point du rectangle à l'aide de la fonction `point`. Cliquer sur un des rectangles. Le numéro $n = cv(i)$ de sa couleur est déterminé à l'aide de la table de conversion, puis la couleur est changée par `color`(n). Il aurait été plus direct de la changer par `vcolor`(i) en utilisant son index.

```
index cv(15)
if resolution
  message "Programme|Basse Résolution"
  stop
endif
locate 4,0
print "--- index numéro    RVB"
for i=0,15
  f_color i
  pbox 0,40+8*i,23,47+8*i
  n=point(10,44+8*i)
  cv(i)=n
  locate 5+i,6
```

```
      print justr$(i,2);justr$(n,6);
      print/h/using "      ###",color(n)
    next i
    hidec
    do
      if mousex in [0,23]
        i=divr(mousey-40,8)
        if i in [2,15]
          n=cv(i)
          j=random($800) and $777
          color(n)=j
          locate 5+i,20
          print/h/using "###",color(n);
        endif
      endif
    loop
```

POINT(x, y)

VPOINT(x, y)

V_fonctions Numéro et Index de couleur

x, y

réels, coordonnées relatives

La fonction `point` (resp `vpoint`) renvoie le numéro (resp l'index) de la couleur du point (x, y).

Exemple (monochrome)

Le tableau d'index*1 `ecr`, défini par `local` ... `access`, donne accès à la mémoire écran. Le point (x, y) de l'écran correspond à l'élément `ecr`$(640y + x)$.

```
    local datai $31303030,256000,xbios(2),$7030020 access
     ecr(255999)
    print point(40,200);vpoint(40,200)
    ecr(640*200+40)=-1
    print point(40,200);vpoint(40,200)
```

Sortie (365 ms)

```
 0  0
 1  1
```

GRAPHMODE n

Variable d'état Mode graphique

n

entier $n \in [1, 4]$

La variable d'état `graphmode` a quatre valeurs possibles qui conditionnent les affichages graphiques :

n	mode graphique
1	remplacé
2	transparent
3	inversé
4	renversé transparent

Exemple (monochrome)

Une boîte 51×82 et la valeur de `graphmode` sont affichées pour chacun des modes graphiques.

```
f_type 3
f_style 8
pbox 0,101,639,142
f_type 2
f_style 9
for i=1,4
  graphmode i
  origin 150*i-100,80
  pbox 0,0,50,81
  text 5,100,"mode="&justl$(graphmode)
next i
```

Sortie (755 ms)

La remarque suivante n'a d'intérêt que pour l'utilisateur du GDOS. En tant que fonction, la valeur `graphmode` est demandée au système (par `vdi(35)`) et concerne la station de travail courante.

Tracé de points et de lignes

L_COLOR i

L_TYPE t

L_WIDTH e

L_BEGIN d

L_END f

Variables d'état Attributs de lignes

i

entier*16 $i \in [0, 15]$, index de la couleur

t

entier*16

t	type du trait
1	continu
2	tiret
3	point
4	point trait
5	trait
6	point point trait
7	défini par l'utilisateur
$t \notin [1, 7]$	nouveau masque

Si $t \notin [1, 7]$, t est considéré comme étant un masque de 16 bits définissant le type de ligne 7.

e

entier*16 $e \in [1, 39]$
C'est la largeur de ligne en pixels (les valeurs paires de e sont traitées comme la valeur impaire $e - 1$).

d et f

entier*16 $d \in [0, 2]$, $e \in [0, 2]$, début et fin de ligne

d ou f	
0	droit
1	flèche
2	rond

Exemple (basse résolution)

Affiche des lignes de diverses couleurs et l'index de couleur `l_color`.

```
for i=0,15
  l_color i
  print "Couleur";l_color
  line 90,8*i+36,190,8*i+36
next i
```

Exemple (monochrome)

L'exemple suivant dresse une table des types des lignes, avec des exemples de styles utilisateur aléatoires. Devant chaque ligne tracée, on écrit son type `l_type`, son épaisseur `l_width` et ses extrémités `l_begin` et `l_end`. Arrêt par appui sur une touche.

```
      print/c/justc$("Styles de ligne",25)
      for i=1,6
        table i,1,0,0,i+4
      next i
      for i=1,3
        table 1,3*i,0,0,i+10
      next i
      table 1,1,1,0,14
      table 1,1,0,1,15
      table 1,9,0,2,16
      do
        for i=17,23
          l_type mods(random(2^16),2^16)'style de ligne aléat
           oire
          table 7,1,0,0,i
        next i
        ift keytest exit
        hidecm
      loop
      stop
table:procedure(a,b,c,d,y)
      l_type a
      l_width b
      l_begin c
      l_end d
      locate y,0
      print a;b;c;d;
      hidec
      line 104,16*y+8,200,16*y+8
      return
```

Ce paragraphe n'a d'intérêt que pour l'utilisateur du GDOS. En tant que fonctions, les valeurs `l_color`, `l_type` et `l_width` sont demandées au système (par `vdi(35)`) et concernent la station de travail courante. Par contre, `l_begin` et `l_end`, dont les valeurs ne sont pas fournies par le VDI, renvoient les dernières valeurs fixées par les commandes `l_begin` et `l_end` (qui peuvent concerner une autre station de travail, si elle a été modifiée).

PLOT x, y {, xi, yi}

Commande Trace un ou plusieurs points

x, y, . . .

réels, coordonnées relatives

La taille des points est e (définie par `l_width` e). Si $e \neq 1$, il faut définir les extrémités rondes par :

```
l_end 2
l_begin 2
```

sinon les points ne sont pas tracés.

Exemple

La commande `plot` trace deux rangées de points de taille 1, puis une rangée de points d'épaisseur 5.

```
for i=0,620,20
  plot i,175,i,195
next i
l_width 5
l_begin 2
l_end 2
for i=10,630,20
  plot i,185
next i
```

Sortie (1480 ms)

LINE [TO] x0, y0 TO x1, y1 { TO x2, y2 }

Commande Trace une ou plusieurs lignes

x0, y0, . . .

réels, coordonnées relatives

Le déterminant `to` peut être remplacé par une virgule. La forme :

```
line to x,y
```

trace la ligne à partir du dernier point utilisé par `line` ou `plot`.

Exemple (monochrome)

Tracé d'un système d'axes et de la courbe :

$$y = f(t) = \frac{\sin t}{t}\cdot$$

La première instruction `line` trace la partie gauche de l'axe des t jusqu'à la flèche marquant l'unité. Le reste de l'axe est tracé par la troisième instruction `line`. La courbe est tracée dans la boucle `for` par une multitude de petits segments de droite, en joignant le point $(at_j,\, bf(t_j))$ à l'extrémité du segment précédent $(at_{j-1},\, bf(t_{j-1}))$. Les valeurs t_j utilisées sont $j/5 = \texttt{i}/50$ pour $j = 1, \ldots, 60$ et les facteurs d'échelle sont $a = 50$ et $b = -100$ (noter la valeur négative).

```
origin 10,270
l_end 1
line -10,0,50,0
line 0,150,0,-100
```

```
    l_end 0
    line 50,0,600,0
    line 0,-100,0,-150
    text 580,-8,"t"
    text 5,-140,"y=sin(t)/t"
    plot 0,-100
    for i=10,600,10
      line,i,-5000*sin(i/50)/i
    next i
```

Sortie (5940 ms)

Remplissages

F_COLOR i

F_TYPE t

F_STYLE s

Variables d'état Attributs de remplissage

F_USER g, f1, f2, ..., fg

Commande Motif de remplissage personnel

i

entier $i \in [0, 15]$, index de couleur

t

entier $t \in [0, 4]$, type de remplissage

s

entier $s \in [1, 24]$, style de remplissage

t	*s*	motif de remplissage
0		vide
1		plein
2	1 à 24	divers motifs
3	1 à 12	diverses hachures
4		utilisateur

g

entier

f1, ..., fg

Motif de remplissage personnel

Exemple

Trace un disque dans une couleur aléatoire d'index `f_color`.

```
f_color random(15)+1
print f_color
pcircle 150,120,40
```

Sortie (605 ms)

Exemple (monochrome)

Le programme suivant crée un motif de remplissage par `f_user`. L'argument g, qui indique le nombre de mots composant le motif, doit être égal à 16 fois le nombre de plans vidéo ($g = 16$ en monochrome). La commande `clip` est utilisée pour limiter l'affichage.

```
f_user 16,-1,-1,-1,-1,-1,-1,0,0,0,0,0,0,-1,0,0,0
f_type 4
origin 320,200
l_type $5555
l_type 7
box 40,115,60,130
pbox 0,0,100,100
clip -320,-200,0,200
pcircle -50,0,70
clip 100,-200,320,200
pcircle 150,0,70
clip 0,0,0,0,0
l_width 3
ellipse 50,10,200,130
```

Sortie (1260 ms)

Exemple

Le programme suivant sort la table de tous les motifs de remplissage. En plus des motifs prédéfinis du système, des motifs aléatoires sont construits par `vdi(112)` (qui est équivalent à `f_user`). La valeur `xbios(4)` est identique à `resolution`, mis à part le cas où `resolution` prend la valeur 3 qui est particulière au Basic. La procédure table(a, b, x, y) sort en x, y un rectangle de type `f_type` $= a$ et de style `f_style` $= b$.

```
print/c/justc$("Types et styles de remplissage",80)
table 0,0,0,5
table 1,0,10,5
b=0
for y=8,14,3
  for x=0,70,10
    b=b+1
```

```
        table 2,b,x,y
      next x
    next y
    b=0
    for y=17,20,3
      for x=0,50,10
        b=b+1
        table 3,b,x,y
      next x
    next y
    for x=20,70,10
      plan=2^(2-xbios(4))'nombre de plans vidéo
      b=random(2^(16*plan*16))'motif de remplissage aléatoi
       re
      pokecba intin,mkz$(b,32*plan)
      vdi §112,16*plan
      table 4,0,x,5
    next x
    stop
table:procedure(a,b,x,y)
    f_color 1
    f_type a
    f_style b
    pbox 8*x+32,16*y+8,8*x+79,16*y+30
    locate y+2,x+5
    print justl$(f_type);",";justl$(f_style);
    return
```

Sortie (3875 ms)

La remarque suivante n'a d'intérêt que pour l'utilisateur du GDOS. En tant que fonctions, les valeurs `f_color`, `f_type` et `f_style` sont demandées au système (par `vdi(37)`) et concernent la station de travail courante.

FILL x, y [, c]

Commande Remplir une surface

x, y

réels, coordonnées relatives du point de départ

c

entier*16 (par défaut $c = -1$)

L'argument c indique l'index de la couleur du bord. La valeur -1 indique que toute couleur différente est prise pour bord.

Exemple

La commande `fill` est utilisée pour remplir la surface entre deux ellipses.

```
    f_type 2
```

```
f_style 7
ellipse 200,120,70,30,0,3600
ellipse 200,120,90,40,0,3600
fill 120,120
```

Sortie (810 ms)

Tracé de formes

Le tracé de formes de bords dépend des attributs de ligne, tandis que celui de formes de surfaces dépend des attributs de remplissage.

BOUNDARY [n]

NOBOUNDARY [n]

Variables d'état Tracé du bord

n

entier $n \in [0, 1]$

La valeur $n = 0$ correspond à l'absence de bord, et $n = 1$ à la présence de bord. Dans le cas des formes de surfaces, le bord est une ligne fine continue, qui est tracée ou non suivant la valeur (identique) de ces variables d'état. Chacune des instructions :

```
noboundary
noboundary 0
boundary 0
```

supprime le tracé des bords. Pour remettre le tracé des bords, on a le choix entre les instructions suivantes :

```
boundary
noboundary 1
boundary 1
```

Exemple

Le programme trace une boîte et un disque, avec et sans bord.

```
     f_style 9
     f_type 2
     bord 150,0
     bord 290,1
     stop
bord:procedure(index x,b)
     origin x,182
     boundary b
```

```
pcircle 50,110,30
pbox 0,0,100,68
return
```

Sortie (725 ms)

Pour l'utilisateur du GDOS seulement, mentionnons que `boundary` et `noboundary` se comportent en tant que fonctions comme `l_begin`, et donc de façon pas entièrement satisfaisante pour la raison que leurs valeurs ne sont pas fournies par le VDI. Elles renvoient la dernière valeur fixée, qui peut être fausse si la station de travail a été changée entre-temps.

BOX x, y, xp, yp

Commande Bord d'un rectangle

PBOX x, y, xp, yp

Commande Rectangle plein.

PIBOX x, y, xp, yp

Commande Rectangle plein sans bord.

RBOX x, y, xp, yp

Commande Bord d'une boîte arrondie

PRBOX x, y, xp, yp

Commande Boîte arrondie pleine.

x, y et xp, yp

réels, coordonnées relatives de deux sommets opposés

Exemple

Le programme utilise les commandes `box`, `pbox`, `pibox`, `rbox` et `prbox`.

```
origin 100,100
rbox 0,0,260,170
box 10,10,50,80
l_type 2
box 10,90,50,160
f_type 3
f_style 12
pbox 60,10,110,160
f_type 2
f_style 22
prbox 120,10,180,160
pibox 190,10,250,160
```

Sortie (735 ms)

CIRCLE x, y, r [, a, b]

Commande Trace un [arc de] cercle

PCIRCLE x, y, r [, a, b]

Commande Remplit un disque ou un secteur

x, y

réels, coordonnées relatives du centre

r

réel, rayon

a, b

réels, angles en 1/10 de degré

Exemple

Trace des cercles concentriques avec la commande `circle`.

```
cursh 0
cls
hidecm
originx 320
originy 200
for I=5,375,5
  circle 0,0,I
next
```

Sortie (4290 ms)

Exemple

Illustre la différence entre les deux commandes suivantes :

```
pcircle x,y,r,0,3600
pcircle x,y,r
```

et le tracé avec et sans bord (sans bord après `noboundary`).

```
origin 100,150
f_type 2
f_style 18
pcircle 0,0,45
pcircle 100,0,45,0,3600
pcircle 200,0,70,2100,3300
noboundary
pcircle 200,100,70,2100,3300
```

Sortie (840 ms)

ELLIPSE x, y, rx, ry [, a, b]

Commande Trace une ellipse ou un arc d'ellipse.

PELLIPSE x, y, rx, ry [, a, b]

Commande Trace une [portion d'] ellipse pleine

x, y

réels, coordonnées relatives du centre

rx, ry

réels, demi-axes

a, b

réels, angles en 1/10 de degré

Exemple

Trace une demi-ellipse, puis une série d'ellipses en utilisant la commande `ellipse`.

```
ellipse 150,180,120,70,0,1800
for i=1,60,2
  ellipse 150,200+i,50+i,33+i
next i
```

Sortie (2010 ms)

Exemple

Tracés de formes elliptiques (bords par `ellipse` et formes pleines par `pellipse`).

```
origin 100,100
ellipse 150,100,50,33
l_width 10
l_begin 2
l_end 1
ellipse 150,100,120,70,1800,3600
f_type 2
f_style 11
pellipse 150,100,40,23
pellipse 150,0,120,68,2300,3300
```

Sortie (885 ms)

Marqueurs

M_COLOR i

M_TYPE t

M_HEIGHT h

Variables d'état Attributs de marqueur

i

entier $i \in [0, 15]$, index de couleur

t

entier $t \in [1, 6]$, type de marqueur

t	marqueur
1	point
2	plus
3	étoile
4	carré
5	croix
6	carreau

h

entier, hauteur du marqueur en pixels

Il n'y a que 8 hauteurs qui sont effectivement possibles.

Les remarques suivantes n'intéressent que l'utilisateur du GDOS. En tant que fonctions, les valeurs `m_color`, `m_type` et `m_heigth` sont demandées au système (par `vdi(36)`) et donnent la valeur retenue pour la station de travail courante. Cependant, `vdi` 36 renvoie (bogue) le type de marqueur diminué de 1, et Basic 1000d ajoute 1 à cette valeur.

MARK x, y { , xi, yi }

Commande Trace un ou plusieurs marqueurs

x, y, . . .

réel, coordonnées relatives

Exemple

La commande `mark` trace les 6 types de marqueurs possibles. Remarquer que la taille du point (type 1) est toujours 1×1.

```
t_height 6
for i=1,6
  for j=1,6
    m_type i
    m_height j*11
    origin 90*i-4,88+j*(j-1)*9
    mark 0,0
    text 5*j,-2,justl$(m_height)
    text 5*j,6,justl$(m_type)
  next j
next i
```

Sortie (2565 ms)

Polyline, Polyfill et Polymark

Ces commandes transmettent les coordonnées des points sous la forme de tableaux. A la différence des autres commandes graphiques, seulement des coordonnées entières sont acceptées.

POLYLINE x(k...), N, p, y(j...), q

Commande Ligne brisée

POLYFILL x(k...), N, p, y(j...), q

Commande Polygone plein

POLYMARK x(k...), N, p, y(j...), q

Commande Marqueurs

N

entier $N \geq 0$, nombre de sommets

x(k...), y(j...)

nomi, coordonnées relatives du premier sommet

p, q

pas entre les éléments x() et y() respectivement

La donnée des coordonnées x des sommets est effectuée par la même méthode que pour les commandes `copy` ou `sort`. Pour la donnée des coordonnées y, on indique seulement le premier élément et le pas. Les tableaux `x` et `y` peuvent être multidimensionnés. Des tableaux de types var ou index sont possibles mais les valeurs non entières ne sont pas admises.

Exemple

La commande `polyline` trace une ligne brisée, la commande `polyfill` remplit un polygone et la commande `polymark` affiche des marques. Les arguments `x(0),8,1` indiquent que les coordonnées x des points sont données par x(0), x(1), ..., x(7).

```
index x(7),y(7)
for i=0,7
  x(i)=cint((10+i)*4.5*cos(i))
  y(i)=cint((10+i)*4.5*sin(i))
next i
f_type 2
f_style 21
origin 80,200
polyline x(0),8,1,y(0),1
origin 240,200
```

```
          polyfill x(0),8,1,y(0),1
          m_type 3
          m_height 11
          polymark x(0),8,1,y(0),1
```

Sortie (1835 ms)

Exemple

Les arguments `X(0),N,2,Y(0),2` indiquent que la commande concerne les N points (X(0), Y(0)), (X(2), Y(2)), ..., (X($2N-2$), Y($2N-2$)).

```
          var X(126),Y(126)
          for N=5,11,2
            for I=0,N
              X(I)=cint(100*cos(2*pi*I/N))
              Y(I)=cint(100*sin(2*pi*I/N))
              X(I+N)=X(I)
              Y(I+N)=Y(I)
            next I
            origin 320,200
            cls
            polyline X(0),N+1,1,Y(0),1
            polyfill X(0),N,2,Y(0),2
            origin 110,200
            polyfill X(0),N,2,Y(0),2
            origin 530,200
            m_type 4
            polymark X(0),N,1,Y(0),1
            polyline X(0),N+1,1,Y(0),1
          next N
```

Sortie (7725 ms)

Textes graphiques

T_COLOR i

T_TYPE t

T_ANGLE a

T_HEIGHT h

T_FONT f

Variables d'état Attributs de textes graphiques

i

entier $i \in [0, 15]$, index de couleur

t

entier*16

Le bit_i de t sélectionne, lorsqu'il est mis, un effet de style.

i	effet
0	gras
1	fin
2	italique
3	souligné
4	éfilé

Par exemple, $t = \%01001$ correspond à l'écriture grasse soulignée.

a

entier*16, angle d'écriture

a	angle d'écriture
0	normal
900	vers le haut
1800	vers la gauche
2700	vers le bas

h

entier*16, hauteur en pixels

Les hauteurs utilisables vont de 4 à 32

h	
4	mini
6	normal couleur
13	normal monochrome
32	maxi

f

entier*16, identificateur de fonte

Ce paragraphe n'a d'intérêt que pour l'utilisateur du GDOS. En standard seule la fonte système (`t_font` vaut 1) est disponible. Sous GDOS, il est possible

de charger d'autres fontes par `vdi(119)` ou `vst_load_fonts` de la bibliothèque STND. Les attributs peuvent alors prendre d'autres valeurs que celles données dans les tables ci-dessus. En tant que fonctions, les valeurs `t_color`, `t_angle`, `t_height` et `t_font` sont demandées au système (par `vdi(38)`) et concernent la station de travail courante. Par contre, comme le VDI ne fournit pas la valeur retenue de `t_type`, la fonction `t_type` renvoie seulement la dernière valeur fixée.

TEXT x, y, ch

Commande Affiche ch en mode graphique

x, y

réels, coordonnées relatives du point d'action

ch

virchaîne

Exemple

La commande `text` écrit en gras souligné.

```
t_type %1001
text 200,150,"texte graphique"
```

Sortie (515 ms)

On trouvera, commande `vdi(-10)`, un exemple de sortie de texte graphique avec justification.

19

Appels système



Basic 1000d permet tous les appels systèmes. Ce chapitre n'est pas une description du système, mais seulement une description de la façon de l'appeler, avec quelques exemples pratiques. Pour plus de détails et des explications sur le système, nous vous renvoyons au livre de Laurent Besle (1986).

GEMDOS

GEMDOS(k {, wli })

V_fonction Appel de la fonction *k* du TRAP #1

k

entier

wli

entier*16 ou *32

La fonction `gemdos` renvoie le contenu du registre D0 (mot long signé). La liste d'arguments {, wli} dépend du numéro *k* de la fonction. Le nombre et la taille (*16 ou *32) des arguments est vérifié suivant le numéro de fonction. L'ordre des arguments est l'ordre inverse d'écriture en assembleur, et le même ordre qu'en C.

Exemple

La fonction $36 du TRAP #1, demande des informations sur le disque. L'appel de cette fonction en assembleur se fait par un programme ressemblant à celui ci-dessous. Le nombre *drive* désigne le lecteur de disques (0 le lecteur de disques par défaut, 1 pour le lecteur A, ...). Le nombre *tampon* est l'adresse d'un tableau de 16 octets où la fonction écrit les informations.

```
MOVE #drive,-(SP)
MOVE.L #tampon,-(SP)
MOVE #$36,-(SP)'numéro de fonction
TRAP #1'appel de la fonction
ADDQ #8,SP'correction de pile
```

Cet appel peut être effectué en Basic 1000d comme suit :

```
index tp(3)
tampon=ptr(tp(0))
drive=0
if gemdos($36,tampon,drive)
  print "disque inactif"
else
  print "nombre de blocs libres";tp(0)
```

```
            print "nb total de blocs du disque";tp(1)
            print "taille d'un secteur";tp(2);" octets"
            print tp(3);" secteurs par bloc"
          endif
```

Sortie (2345 ms)

```
nombre de blocs libres  519
nb total de blocs du disque  711
taille d'un secteur  512 octets
  2 secteurs par bloc
```

Table des fonctions

Les numéros de fonctions permis sont en hexadécimal. Les numéros 0 et $20 sont interdits et les numéros 31 et 4C déconseillés. On a indiqué si des commandes ou fonctions du Basic 1000d sont analogues. Cependant les différences peuvent être très importantes. La liste d'appel est précisée, w indiquant un entier*16 et l un entier*32.

1

Analogue à `keyget`.

2 w

Comme `print chr$(w)`, mais les caractères écrits ne sont pas réutilisables. Par exemple la sonnette s'obtient par :

```
    ift gemdos(2,7)
```

3

Comme `inp`(1), lit un octet sur l'entrée RS 232 (attente infinie).

4 w

Comme `out 1,w`, émet l'octet $w \in [0, 255]$ vers RS 232.

5 w

Comme `lprint chrp$(w);`, émet l'octet w vers la sortie parallèle.

6 w

Analogue à `keytest` ou `inp`(2) (si $w =$ $FF) ou `print chr$(w)` sinon

7

Analogue à `keyget`.

8

Analogue à `keyget`.

9 l

Analogue à `print` c$ où $l =$ `ptr`(c$), mais les caractères écrits ne sont pas réutilisables.

A l

Analogue à `input`.

B

Analogue à `inp?`(2), état du tampon clavier.

E w

Fixe le lecteur de disques par défaut.

La fonction renvoie la somme $\sum 2^i$ sur les lecteurs i actifs ($i = 0$ disque A, $i = 1$ disque B, …, $i = 15$ disque O). Cette fonction est utilisée par la variable d'état `chdrive` du Basic. Voici un exemple pédagogique, où la procédure `chdrive_bis` simule la commande `chdrive`. L'argument doit être une lettre de A à O qui indique le lecteur.

```
          chdrive_bis A
          print chdrive
          stop
chdrive_bis:ift gemdos($e,asc("@1")-1 and $F)
          return
```

Sortie (25 ms)

```
   1
```

10

Etat de l'écran, renvoie −1.

11

Analogue à `out?`(0), état de l'imprimante (renvoie −1 si elle est prête, et 0 sinon). La commande `lprint` appelle cette fonction.

12

Analogue à `inp?`(1), état du tampon RS 232 (entrée) (renvoie −1 si le tampon contient des octets, 0 si le tampon est vide).

13

Analogue à `out?`(1), demande l'état du tampon RS 232 (sortie) (renvoie −1 si le tampon peut accepter des octets, 0 sinon).

19

Analogue à `chdrive`−1, lecteur de disques par défaut (renvoie un nombre de 0 (lecteur A) à 15 (lecteur O).

1A l

Donne un tampon 44 octets pour opérations disques (usage déconseillé).

2A

Renvoie la date codée dans le mot `w`.

bits		
0–4	`modr(w,32)`	jour $\in [1,31]$
5–8	`modr(divr(w,32),16)`	mois $\in [1,12]$

Voir l'exemple `gemdos` 2D.

2B w

Fixe la date. Voir l'exemple `gemdos` 2D.

2C

Renvoie l'heure codée sur le mot `w`.

bits		
0–4	`modr(w,32)*2`	secondes (valeurs paires de 0 à 58)
5–10	`modr(divr(w,32),64)`	minutes (0 à 59)
11–15	`divr(w,2^11)`	heures (0 à 23)

Voir l'exemple `gemdos` 2D.

2D w

Fixe l'heure

Exemple

L'exemple, purement pédagogique, simule les variables d'état `date$` et `time$` du Basic. La procédure `shorloge` règle l'horloge et la fonction `horloge` renvoie l'heure sous la forme “hh:mm:ss”. La procédure `sdate` fixe la date et la fonction `rdate` renvoie la date sous la forme “jj.mm.aaaa”.

```
         push$ date$,time$
         sdate 24,7,88
         shorloge 11,25,20
         print horloge;"   ";rdate
         time$ pop$
         date$ pop$
         stop
shorloge:procedure(h,m,s)
         ift gemdos($2D,mods(h*$800+m*$20+int(s/2),2^16))
         return
   sdate:procedure(jour,mois,an)
         if an>§1980
           an=an-§1980
         else
           an=modr(an-§80,§100)
         endif
         ift gemdos($2B,mods(jour+mois*$20+an*$200,$10000))
         return
 horloge:function$
         local char heure,minute,seconde index i
         i=gemdos($2C)
         heure=(i and $F800)/$800
```

```
          minute=(i and $7E0)/32
          seconde=(i and $1F)*2
          value=date_f(heure)&":"&date_f(minute)&":"&date_f(secon
           de)
          return
    rdate:function$
          local char jour,mois,an
          jour=gemdos($2A) and $1F
          mois=(gemdos($2A) and $1E0)/32
          an=1980+(gemdos($2A) and $FE00)/512
          value=date_f(jour)&"."&date_f(mois)&"."&justr$(an)
          return
   date_f:function$
          value=right$(chr$($30)&justr$(@1),2)
          return
```

Sortie (195 ms)

```
 11:25:20   24.07.1988
```

2F

Renvoie l'adresse du tampon 44 octets (disque). Voir `gemdos` 4F

30

Renvoie le numéro de version du `gemdos`.

31 l, w

Libère mémoire (ne pas utiliser).

36 l, w

Information sur disque (voir exemple en début de la section `gemdos`)

39 l

Création d'un répertoire.

Exemple

Les procédures `mkdir_bis` et `rmdir_bis` suivantes sont identiques aux commandes `mkdir` et `rmdir` du Basic.

```
          mkdir_bis "A:\REP"
          print files$("A:\")
          rmdir_bis "A:\REP"
          stop
mkdir_bis:local datac @1&chr$(0) char c
          ift gemdos($39,ptr(c))=0 return
          print "Répertoire non créé"
          stop
rmdir_bis:local datac @1&chr$(0) char c
          ift gemdos($3A,ptr(c))=0 return
          ift gemdos($3A,ptr(c))=0 return
```

```
          print "Répertoire non effacé"
          stop
```

Sortie (6845 ms)

```
 [ A: ] A:\*.*    716800 octets libres
    325 MATH1.Z     ,   130 HELPEX.Z    ,     0 BAS1.PRG    ,   206 T.Z
      0+REP
```

3A l

Efface un répertoire. Voir exemple `gemdos` 39

3B l

Fixe le répertoire courant.

Exemple

La commande `chdir` du Basic a le même effet, les messages mis à part, et la même syntaxe que la procédure `chdir_bis` suivante. Noter que la chaîne doit être suivie de `chr$`(0).

```
          chdir_bis "A:\"
          chdir_bis "C:\pagaille"
          stop
chdir_bis:procedure(char c)
          c=c&chr$(0)
          select gemdos($3B,ptr(c))
          case=0
            print "Répertoire sélectionné ";c
          case others
            print "Répertoire inconnu ";c
          endselect
          return
```

Sortie (100 ms)

```
 Répertoire sélectionné A:\
 Répertoire inconnu C:\pagaille
```

3C l, w

Création d'un fichier disque. Les attributs (voir `files$`) sont w, et l est l'adresse du nom.

3D l, w

Ouverture d'un fichier existant sur disque ($w = 0$ lecture seule, $w = 1$ écriture seule, $w = 2$ lecture/écriture). l est l'adresse du nom. La fonction renvoie l'identificateur du fichier (ou un nombre négatif si erreur).

3E w

Ferme le fichier d'identificateur w.

3F w, N, l

Lit N (entier*32) octets du fichier d'identificateur w et les implante à partir de l'adresse l. On peut donner $N < 0$ pour lire tout le fichier. Renvoie le nombre d'octets lus ou une valeur négative si erreur.

40 w, N, l

Envoie les N (long) octets en l dans le fichier w. Renvoie le nombre d'octets transmis ou une valeur négative si erreur.

41 l

Efface un fichier disque.

Exemple

L'exemple suivant, crée la procédure `kill_bis`, identique à la commande `kill` du Basic.

```
         save$ "aaaaaaaa.aaa","?"
         print files$
         kill_bis "aaaaaaaa.aaa"
         stop
kill_bis:procedure(char c)
         ift gemdos($41,ptr(c)) print "Fichier ",c," pas trouvé"
         return
```

42 l, w, w

Déplace le pointeur d'un fichier disque.

43 l, w, w

Fixe ou renvoie les attributs d'un fichier disque.

45 w

Renvoie un autre identificateur du fichier w.

46 w, w

Redéfinit un identificateur de fichier.

47 l, w

Renvoie le répertoire courant ($w = 0$ disque courant, $w = 1$ disque A, $\ldots$, $w = 15$ disque O). l est l'adresse d'un tampon de 64 octets où la fonction écrit le nom du répertoire. La fonction est utilisée par `dir$` du Basic.

Exemple

Voici une procédure qui a le même effet que `print dir$`(n).

```
         printdir 0
         printdir 3
         stop
printdir:procedure
         local char dn
         dn=chr$(0,$64)
```

```
print "Disque @1";
if gemdos($47,ptr(dn),@1)
  print " Non trouvé"
else
  print f;peekz$(ptr(dn))
endif
return
```

48 l

Demande d'allocation mémoire.

49 l

Libération de mémoire.

4A w, l, l'

Rétrécit la taille d'un bloc alloué. w est arbitraire, l l'adresse de base de l'allocation, et l' la nouvelle taille du bloc, qui doit être inférieure à la précédente.

4B w, l, l, l

Exécute un programme.

4C w

Termine un programme.

4E l, w

Recherche un fichier. w indique les attributs (voir `files$`) recherchés et l est l'adresse du nom (pointe `A:*.Z` par exemple). Voir `gemdos` 4F.

4F

Recherche un fichier.

Exemple

L'exemple suivant liste les fichiers du disque A. Les fonctions $4E et $4F renvoient dans le tampon en x les informations suivantes sur le fichier :

octets	
0 à 20	réservé au système
21	attribut
22 et 23	heure
24 et 25	date
26 à 29	longueur
30 à 43	nom

```
char c
c="A:\*.*"æ
if gemdos($4E,ptr(c),1)
  print "Pas de fichier"
```

```
        else
          print "Fichier      longueur"
          repeat
            x=gemdos($2F)
            print justl$(peekz$(x+30),15);justr$(peekls(x+26),6
             )
          until gemdos($4F)
        endif
```

Sortie (4605 ms)

```
Fichier      longueur
DESKTOP.INF       477
T.Z               206
...
```

56 w, l, l

Change le nom d'un fichier.

Exemple

La procédure `rename` suivante est identique à la commande `name` ... `as` du Basic.

```
        rename "old.old","new.new"
        stop
 rename:local datac @1&chr$(0),@2&chr$(0) char a,b
        ift gemdos($56,0,ptr(a),ptr(b))=0 return
        print "Erreur rename Ancien nom "&a&" Nouveau nom "&b
        stop
```

57 l, w, w

Date de création d'un fichier.

BIOS

BIOS(k {, wli })

V_fonction Appel de la fonction *k* du TRAP #13

Les remarques sur la liste des arguments vues dans `gemdos` s'appliquent à la fonction `bios`.

Table des fonctions

0 l

Informations sur la mémoire.

1 w

Analogue à `inp?`(w), état du périphérique w (entrée).

w	périphérique
0	imprimante (port parallèle)
1	RS 232 (série)
2	clavier et console
3	MIDI
4	processeur clavier (inutilisable)
5	écran

Exemple

Les fonctions `inp_?` et `out_?`, qui simulent `inp?` et `out?` du Basic, demandent l'état du périphérique w en entrée ou sortie, et renvoient -1 si le périphérique est prêt et 0 sinon. La fonction `inp_bis` qui attend un octet sur le périphérique w (l'attente est infinie) simule la fonction `inp`. La procédure `out_bis` qui envoie des octets au périphérique w simule la commande `out`.

Le programme suivant écrit la table des codes ASCII, puis les codes des touches appuyées, jusqu'à l'appui sur Escape.

```
        print /c/
        for i=0,255
          out_bis 5,32,i
          ift modr(i,16)=15 print
        next i
        do
          i=inp_bis(2)
          ift i=$1b stop
          print /h/ i;
        loop
        stop
inp_bis:function(x)
        local var i
        i=bios(2,x)
        value=i and $FFFF
        ift x<>2 or value return
        value=divr(i,$10000) or $80
        return
  inp_?:function
        value=bios(1,@1)
        return
out_bis:local datai @1,@0 index x,a
        ift a<2 return
```

```
        for a=2,a
          ift bios(3,x,modr(@a,$100))
        next a
        return
  out_?:function
        value=bios(8,@1)
        return
```

2 w

Attend un octet sur le périphérique *w*. Voir l'exemple `bios` 1.

3 w, b

Envoie l'octet *b* sur le périphérique *w*. Voir l'exemple `bios` 1.

4 w, l, w, w, w

Lecture/écriture de secteurs sur disque.

5 w, l

Fixe/renvoie un vecteur d'interruption.

6

Période du compteur.

7 w

Bloc de paramètres du disque *w*.

8 w

Etat du périphérique *w* (sortie). Voir l'exemple `bios` 1.

9 w

Teste si le disque a été changé.

A

Lecteurs de disques actifs.

Exemple

```
D0=bios($A)
for I=0, 15
  M=modr(D0,2)
  D0=divr(D0,2)
  ift M print "disque ";chr$(I+$41);" actif"
next
```

Sortie (185 ms)

```
disque A actif
disque B actif
```

B w

Touches spéciales du clavier.

XBIOS

XBIOS(k {, wli })

V_fonction Appel de la fonction *k* du TRAP #14
Les remarques sur la liste des arguments vues dans `gemdos` s'appliquent à la fonction `xbios`.

Table des fonctions

0 w, l, l
Initialisation : souris

2
Base physique de l'écran

3
Base logique de l'écran

4
Résolution (analogue à `resolution`)

5 l, l, w
Fixe l'écran et la résolution

6 l
Fixe les couleurs

7 w, w
Fixe une couleur (analogue à `color`)

8 l, l, w, w, w, w, w
Lecture de secteurs sur disque

9 l, l, w, w, w, w, w
Ecriture de secteurs sur disque

A l, l, w, w, w, w, w, l, w
Formatage d'une piste

C w, l
Envoi d'une chaîne vers l'interface MIDI

D w, l
Interruption du MFP

E w
Paramètres d'un tampon d'entrée

F w, w, w, w, w, w

Configuration du RS 232

10 l, l, l

Tables de décodage du clavier

11

Analogue à `random`(2^{25}).

12 l, l, w, w

Tampon bootsector

13 l, l, w, w, w, w, w

Vérification de secteurs disque

14

Copie écran sur imprimante

Exemple

La procédure copieecran est équivalente à `hardcopy`.

```
    copieecran
    stop
copieecran:ift xbios($14)
    return
```

15 w, w

Attributs du curseur

16 l

Fixe date et heure

17

Date et heure

18

Clavier système

19 w, l

Commandes du 6301

1A w

Désactive une interruption du 68901

1B w

Active une interruption du 68901

1C w, w

Lit/écrit un registre son

1D w

Positionne le bit_w du port A à 0

1E w

Positionne le bit_w du port A à 1

1F w, w, w, l

Initialise un timer du 68901

20 l

Voir `music`

21 w

Fixe/renvoie la configuration imprimante

22

Renvoie le descripteur de vecteurs clavier

23 w, w'

Lit/écrit les délais d'autorépétition des touches. w est le premier délai et w' le délai suivant en 20 ms (-1 pour lire).

24 l

Copie d'écran

25

Attend une interruption verticale

26 l

Exécution en mode TRAP

27

Suppression de l'AES

40 w

Blitter

w	
-1	demande la configuration du blitter
0	éteint le blitter (transferts par logiciel)
1	active le blitter

En retour, on obtient la configuration antérieure. Les bits 0 et 1 sont significatifs.

bit_0	Transferts par	bit_1	
0	logiciel	0	Pas de blitter
1	blitter	1	Blitter implanté

Appels 68000

CALL adressepaire {, argwl}

Commande Appel 68000

argwl

```
|   entier*16
| W:entier*16
| L:entier*32
```

Désignent des arguments passés par la pile SP.

REGISTER

Constante

REGISTER(i)

Index*32

i

entier $i \in [0, 14]$

La constante `register` est l'adresse d'une table de 15 mots longs servant au passage des registres D0–D7/A0–A6 avant et après l'appel. On peut lire et écrire dans cette table par `register`(i), comme si `register` était un tableau d'index*32. Les 8 premiers mots longs de ce tableau, `register`(i) pour $i \in [0, 7]$, correspondent aux registres Di. Les 7 mots longs suivants, `register`($i+8$) pour $i \in [0, 6]$, correspondent à Ai.

Exemple

Le programme suivant, en assembleur, implante les cubes des entiers x de 0 à d0 comme des mots longs à partir de l'adresse a0. Le registre d0 sert de compteur, et sur la ligne `cube`, d1 vaut $3x$, d2 vaut $3x^2$ et d3 vaut x^3, x augmentant d'une unité à chaque passage sur cette ligne.

```
     moveq#0,d3
     moveq#0,d2
     moveq#0,d1
cube:move.l d3,(a0)+
     add.l d2,d3
     add.l d1,d3
     addq.l #1,d3
     add.l d1,d2
     add.l d1,d2
     addq.l #3,d2
     addq.l #3,d1
```

```
dbra d0,cube
rts
```

Ce programme, relogeable, qui occupe 28 octets est implanté dans la chaîne machine$, puis exécuté avec le registre a0 initialisé sur l'adresse du tableau d'index cube, et le registre d0 sur 1290 qui correspond au plus grand cube entier*32. On aurait pu initialiser les registres par :

```
pokels register,1290,0,0,0,0,0,0,0,ptr(cube(0))
```

Le `call` est équivalent à :

```
for i=0,1290
  cube(i)=i^3
next i
```

mais alors que la boucle en Basic prend 6 s, le `call` est effectué en 15 ms.

```
index cube(1290)
n=$76007400720020c3d682d6815283d481d4815682568151c8ffee
 4e75
machine$=mkz$(n,28)
register(0)=1290
register(8)=ptr(cube(0))
call ptr(machine$)
print cube(1234);1234^3
```

Sortie (105 ms)

```
1879080904  1879080904
```

Exemple

La liste placée après l'adresse du programme machine dans `call` permet de transmettre des mots et des mots longs par la pile. Le programme en assembleur suivant recherche une chaîne dans une zone mémoire. Il utilise comme entrée les données dans la pile :

(SP)	L	adresse de retour du sous-programme
4(SP)	W	longueur de la chaîne
6(SP)	L	adresse de la chaîne
10(SP)	L	fin de la zone
14(SP)	L	début de la zone

En sortie, le programme renvoie dans le registre D0 l'adresse où la chaîne a été trouvée, ou bien 0 si elle est absente. Le retour du sous-programme peut se faire par "RTS", mais dans cet exemple, le retour se fait en modifiant le pointeur de pile SP. Cela est licite, les programmes appelés par le Basic peuvent modifier tous les registres, y compris SP et SR. En entrée d'un programme appelé par `call`, le processeur est en mode superviseur (ce mode peut être modifié au retour).

```
movem.l 6(sp),a0-a2
move 4(sp),d3
```

```
        sub d3,a1
        move.b (a0)+,d1
        subq#1,d3
        moveq#0,d0
  hunt1:cmp.l a1,a2
        bge.s hunt4
        cmp.b (a2)+,d1
        bne.s hunt1
        move.l a0,a3
        move.l a2,a4
        move d3,d2
        bra.s hunt3
  hunt2:cmpm.b (a3)+,(a4)+
        bne.s hunt1
  hunt3:dbra d2,hunt2
        subq#1,a2
        move.l a2,d0
  hunt4:move.l (sp)+,a0
        jmp (a0)
```

Voici maintenant un programme en Basic qui appelle le programme en assembleur.

```
        a=-1
        do
          a=hunt(a+1 TO xbios(2),"hunt")
          ift a=0 exit
          print a;
        loop
        stop
   hunt:function
        push instrk(lower$(arg$(1)),"to")
        local datai val(left$(arg$(1),stack(0)-1)),val(mid$(arg
         $(1),pop+3)) datac @2f index b,e,a char c$
        index*16 hunt.prg(24)
        a=ptr(hunt.prg(0))
        if hunt.prg(24)<>$4ed0
          pokel a,$4cef0700,$0006362f,$000492c3,$12185343
          pokel a+16,$7000b5c9,$6c18b21a,$66f82648,$284a3403
          pokel a+32,$6004b90b,$66ec51ca,$fffa534a,$200a205f
          pokew a+48,$4ed0
        endif
        call a,l:b,l:e,l:ptr(c$),len(c$)
        value=register(0)
        return
```

Sortie (8 s)

```
6365  6389  ...  960780  960809
```

La fonction `hunt(a TO b, c$)` recherche la chaîne c$ dans la zone mémoire $[a, b[$. Le décodage de `a TO b` est effectué avec l'aide de la pile du Basic, pour éviter toutes interférences avec le programme appelant. Cette méthode est préférable à un programme de décodage comme :

```
hunt:function
     local index x,b,e
     nodistingo
     x=instrk(arg$(1),"to")
     b=val(left$(arg$(1),x-1))
     ...
```

D'une part l'utilisation de `nodistingo`, au lieu de transformer `arg$`(1) par `lower$` peut modifier l'option du programme principal. D'autre part, si par malheur l'appel de la fonction est quelque chose comme (avec une variable `b`) :

```
hunt(b TO c,"xx")
```

le décodage prend `b` pour l'index local (=0) et non pour le `b` du programme appelant.

Le programme machine est installé dans le tableau d'index hunt.prg, seulement lors du premier appel de la fonction `hunt`. Le premier argument de la liste du `call` sera le plus profond dans la pile SP.

CALLA k

CALLF k

Commandes Appel lignes A et F

k

entier $k \in [0, 2^{12}[$. La commande `calla` k exécute l'instruction de code $\$A000 + k$. Les seize valeurs, $k \in [0, 15]$, correspondent à des fonctions graphiques. La commande `callf` k exécute l'instruction de code $\$F000 + k$.

Exemple

```
calla 0
D0=register(0)
A1=register(9)
print "adresse des pointeurs fontes ="; A1
print/h/"adresse du bloc des variables de la ligne A=";
  D0
```

VDI et GDOS

GDOS?

V_fonction Teste si le programme GDOS est résident
La fonction `gdos?` renvoie −1 (vrai) si oui, et 0 (faux) sinon.

Exemple

Les instructions suivantes peuvent se placer en tête d'un programme utilisant GDOS.

```
if not gdos?
  message "Programme GDOS|non résident"
  stop
endif
```

V_H0

V_fonction Identificateur de la station VDI du Basic.

V_H

Variable d'état Identificateur de la station VDI.
La fonction `v_h0` renvoie l'identificateur de la station virtuelle (écran) ouverte par le Basic. Initialement, `v_h` vaut également `v_h0`. Lorsque GDOS est résident, il est possible d'ouvrir d'autres stations VDI (voir les commandes `vdi(1)` et `vdi(100)` ci-dessous, et `v_opnwk` et `v_opnvwk` de la bibliothèque STND). En donnant à `v_h` l'identificateur d'une telle station, on déroute tous les appels VDI vers cette station. Ce déroutage est aussi effectuée dans toutes les commandes qui utilisent le VDI (par exemple `circle` et `graphmode`).

Les variables d'état graphiques, comme par exemple `l_color`, utilisées en tant que commandes, agissent également sur la station `v_h`. Lorsque elles sont utilisées en tant que fonction, elles demandent leurs valeurs si possible au VDI, et renvoient donc la valeur effectivement retenue pour la station `v_h`. Cependant, il n'existe pas d'appel VDI fournissant la valeur des variables d'état suivantes : `boundary`, `noboundary`, `t_type`, `l_begin`, `l_end` et `m_type`. Dans ce cas, le Basic a mémorisé la dernière valeur assignée à la variable d'état, et c'est cette valeur qu'il renvoie. Elle ne correspond donc pas nécessairement à la valeur de la station `v_h`.

Pour des exemples de `v_h`, on se reportera à la section GDOS de la bibliothèque STND.

WORK_OUT

WORK_OUT(i)

V_fonction

i

entier dans [0,62]

La forme `work_out` renvoie l'adresse où se trouvent les informations retournées par la commande implicite `vdi(100)` lors de l'ouverture de la station de travail par le Basic. Ces informations, qui sont des entiers*16, peuvent être lues par `work_out`(i) ou `peekw(work_out+2*i)`. Les valeurs $i \in [0, 44]$ (resp $i \in [45, 56]$)

sont les sorties `intout`(i) (resp `ptsout`($i-45$)) de la commande `vdi 100`. Pour l'interprétation de ces valeurs, voir le livre de L Besle (1986), p152 (où elles sont désignées par arg_out[i]).

Les valeurs $i \in [57, 62]$ sont des variables internes du Basic. Les valeurs normales en résolution 2 sont notées n.

i		n
57	valeur maximum de `cursl`	24
58	`cursh`	4
59	valeur maximum de `cursl`	24
60	`cursl`	
61	valeur maximum de `cursc`	79
62	`cursc`	

On peut obtenir des effets spéciaux d'impression et entrée en modifiant par `pokew` les valeurs `work_out`(57) et `work_out`(59) (qui doivent être égales et au plus égales à 49) ou la valeur `work_out`(61) (qui doit être égale à -1 modulo 4 et au plus égale à 79). En effet, le canal "VBS" du Basic est paramétré par ces éléments 57, 59 et 61. Ces valeurs sont réinitialisées lors d'un changement de résolution. Les éléments 0–56 de la table, par contre, ne sont pas utilisés par le Basic, et les modifier n'a pas d'effet.

CONTRL

INTIN

PTSIN

INTOUT

PTSOUT

Constantes

CONTRL(k)

INTIN(i)

PTSIN(j)

INTOUT(i)

PTSOUT(j)

Index*16

k

entier $k \in [0, 11]$

i, j

entier $i \in [0, 127]$, $j \in [0, 255]$

En tant que constantes, ce sont les adresses de tables de 24, 256 ou 512 octets permettant le passage des paramètres au VDI. Ces tables peuvent être initialisées ou lues en utilisant les formes indicées (correspondant à des mots de 2 octets). Ainsi :

```
intin(0)=7
ptsin(4)=x
ptsin(5)=y
v=ptsout(0)
w=intout(1)
```

équivaut à :

```
pokews intin,7
pokews ptsin+8,x,y
v=peekws(ptsout)
w=peekws(intout+2)
```

VDI

VDI [#id,] k [, n] {, ptsi} {, inj} [, S]

VDIR

VDIR [#id,] k [, n] {, ptxi, ptyi} {, inj} [, S]

Commandes Appel VDI

VDIF(...)

VDIRF(...)

V_fonctions Appel VDI

id

entier*16, identificateur de la station de travail

k

entier $k \in [-10, 131]$, numéro de fonction

n, ptsi, inj

entier*16

ptxi, ptyi

réels, coordonnées relatives

S

exprchaîne

Les appels du VDI peuvent être effectué par les commandes `vdi` et `vdir`, et par les fonctions `vdif` et `vdirf`. Les arguments de `vdi` et `vdif` d'une part, de `vdir` et `vdirf` d'autre part, sont identiques. Les fonctions renvoient en sortie la valeur de `intin`(0). Si aucun argument n'est donné, la commande ou fonction effectue un appel VDI brut sans initialiser les tableaux. Il faut donc initialiser les tableaux `contrl`, `ptsin` et `intin` avant l'appel brut VDI.

Δ Les tableaux `contrl`, `intin`, etc. sont également utilisés par le Basic. Le bloc de paramètres, en `systab` + 2, qui contient les adresses de ces tableaux peut être modifié. Dans ce cas, le Basic utilisera également pour les appels internes ces nouveaux tableaux de paramètres. Noter cependant que le retour à l'éditeur remet les tableaux d'origine du Basic. La modification du bloc de paramètres est donc impossible en mode direct. Les mots clefs `contrl`, `intin`, etc. se rapportent toujours aux tableaux d'origine du Basic et non aux tableaux implantés dans le bloc de paramètres.

Si le numéro k de la commande `vdi` est donné, Basic 1000d se charge de remplir `contrl`, `ptsin` et `intin` suivant la liste d'arguments, qui dépend de la valeur de k. Cette liste peut être donnée de façon incomplète, les variables non rentrées n'étant pas initialisées.

Basic 1000d remplit les 6 valeurs `contrl`(0) à `contrl`(5) selon le numéro de fonction k, sauf pour les fonctions k où `contrl`(1) ou `contrl`(3) peut être variable. Dans ce cas la valeur variable mise est l'argument suivant n. La valeur `contrl`(6) est initialisée avec la variable d'état `v_h`. C'est l'identificateur de la station de travail. Pour l'utilisation de GDOS, on peut soit modifier la valeur de `v_h`, soit indiquer (par #id avant les autres arguments) un autre identificateur. Ensuite les arguments pts1, pts2, ... sont utilisés pour remplir `ptsin`. Dans le cas de `vdir`, ce remplissage est fait à partir des coordonnées relatives ptx1, pty1, ... Autrement dit `ptsin` est initialisé avec `originx` + `cint`(ptx1), `originy` + `cint`(pty1), ... Les arguments inj et la chaîne S (qui est placée pour les fonctions $k = 8$, 116 et -10) servent à remplir `intin`.

Les valeurs $k \in [-10, -1]$ correspondent à la fonction 11, sous-fonction $|k|$ du VDI. Les sorties du VDI sont lisibles dans les tables `intout` et `ptsout`.

Exemple

L'appel de la fonction 32 change le mode graphique. Ainsi :

```
vdi 32,3
```

est équivalent à :

```
graphmode 3
```

Lorsque cet appel est effectué par `vdif` on obtient en retour le mode effectivement sélectionné.

```
print vdif(32,2);graphmode
```

Sortie (25 ms)

```
 2  2
```

Réentrance du VDI/AES

Les commandes `vdi` et `aes`, comme toutes les commandes du Basic 1000d, sont réentrantes (peuvent s'appeler elles-mêmes). Ainsi on peut utiliser :

```
coul=vdif(105,fx(j),fy(t))
```

où les fonctions en Basic fx(j) et fy(t) appellent aussi des commandes du VDI. Par contre, l'appel de la forme :

```
pokew ptsin,fx(j),fy(t)
coul=vdif(105)
```

est exécuté de la façon suivante. fx(j) est calculé et implanté en `ptsin`. Ensuite fy(t) est calculé, mais si fy(t) appelle des commandes du VDI, que se soit directement par la commande `vdi` ou indirectement par des commandes du Basic comme `circle`, alors la valeur fx(j) en `ptsin` sera détruite.

Il faut donc prendre des précautions dans les appels multilignes du VDI et de l'AES si entre l'implantation de la première valeur et l'appel proprement dit d'autres appels du VDI/AES peuvent intervenir. Une façon de procéder est d'utiliser des sous-programmes qui ne détruisent pas les tableaux de paramètres `intin`, `ptsin`, ... C'est le cas de la fonction `cpoint`(x, y) suivante :

```
cpoint:function(x,y)
       push$ screen$
       value=point(x,y)
       rscreen pop$
       return
```

qui renvoie l'index de la couleur du point (x, y). Elle utilise `screen$/rscreen` pour conserver les paramètres du VDI.

Exemple VDI et ligne A

Tracé d'un camembert. Cet exemple, purement pédagogique, pourrait être traité plus simplement par les commandes graphiques du Basic.

```
vdi 23,2'type de remplissage
vdi 3 'vide l'écran
vdi -2,160,100,0,0,0,0,100,0,0,3600 'trace cercle
calla 0'appel de la ligne A
A0=register(0)'bloc de variables de la ligne A
pokew A0+$26,160,100 'centre du cercle
for I=0,8'boucle sur 8 rayons
  pokew A0+$2A,160+cint(cos(pi/8*I)*100)'extrémité du r
   ayon
  pokew A0+$2C,100+cint(sin(pi/8*I)*100)'extrémité du r
   ayon
  calla 3'trace le rayon
next
for I=0,8'boucle sur 8 secteurs
  vdi 25,I+3 'met couleur de remplissage
  vdi 24,9+I 'met style de remplissage
  intin(0)=1
  'commande vdi 103,x,y,c avec remplissage manuel
  ptsin(0)=160+cint(cos(pi/8*(I+.5))*50)
  ptsin(1)=100+cint(sin(pi/8*(I+.5))*45)'point dans le
   secteur
  vdi 103
next
```

Sortie (4090 ms)

Table des commandes du VDI

Dans la table des commandes VDI suivante, la liste maximum d'arguments est donnée pour chaque valeur du numéro de la fonction (certains arguments doivent être mis égaux à 0, ce qui est indiqué par un 0). La référence g107 renvoie à la page 107 de G Szczepanowski (1985), et la référence b158 renvoie à la page 158 de L Besle (1986).

-10, x, y, dx, 0, a, b, S

Affiche la chaîne S avec justification g125 b172

x, y

point d'affichage

dx

longueur d'affichage en pixels

a, b

espace les mots (si $a \neq 0$) et caractères (si $b \neq 0$)
`intin`(2) à `intin`(n) et `contrl`(3) sont mis suivant la chaîne S.

Exemple

On écrit divers textes graphiques. La procédure `deftext` permet de modifier les attributs de texte.

```
        char c
        cls
        hidem
        for i=1,10
          j=2^(i-2)
          ift i=1 j=0
          ift i>6 j=random(32)
          c="j=",justr$(j,2),"  "
          ift j and 1 c=c&"gras "
          ift j and 2 c=c&"brillant "
          ift j and 4 c=c&"italique "
          ift j and 8 c=c&"souligné "
          ift j and 16 c=c&"esquissé"
          ift j=0 c=c&"normal"
          deftext j,0,13
          vdi -10,0,88+18*i,10*len(c),0,1,0,c
        next i
        deftext 0,0,6
        vdi -10,0,290,100,0,1,1,"Texte 6"
        deftext 0,2700,32
        vdi -10,400,88,200,0,1,1,"Texte 32"
        stop
deftext:procedure(t,a,h)
```

```
t_type t
t_angle a
t_height h
return
```

Sortie (1195 ms)

-9, x, y, xp, yp

Rectangle arrondi plein g125 b158

Identique à `prbox x,y,xp,yp`.

-8, x, y, xp, yp

Rectangle arrondi g123 b158

Identique à `rbox x,y,xp,yp`.

-7, x, y, dx, dy, a, b

Portion d'ellipse g118 b160

Identique à `pellipse x,y,dx,dy,a,b`.

-6, x, y, dx, dy, a, b

Arc d'ellipse g116 b160

Identique à `ellipse x,y,dx,dy,a,b`.

-5, x, y, dx, dy

Ellipse pleine g121 b160

Identique à `pellipse x,y,dx,dy`.

-4, x, y, 0, 0, r, 0

Disque g114 b159

Identique à `pcircle x,y,r`.

-3, x, y, 0, 0, 0, 0, r, 0, a, b

Secteur g111 b159

Identique à `pcircle x,y,r,a,b`.

-2, x, y, 0, 0, 0, 0, r, 0, a, b

Arc de cercle g108 b159

Identique à `circle x,y,r,a,b`.

-1, x, y, xp, yp

Rectangle plein g106 b158

Identique à `pbox x,y,xp,yp`.

1, a0, ..., a10

Ouvre une station de travail g71 b150

Voir `v_opnwk` de la bibliothèque STND.

2

Ferme station de travail g77 b151

Voir `v_clswk` de la bibliothèque STND.

3

Vide la station de travail g83 b154

Voir `v_clrwk` de la bibliothèque STND. Lorsque la station de travail est un écran, la fonction efface l'écran.

4

Mise à jour g84

Voir `v_updwk` de la bibliothèque STND.

5

Editeur du VDI b175–8 b180 b188

Basic 1000d met `contrl`(1)=0 et `contrl`(3)=0. Les sous-fonctions doivent être appelées en pokant dans `contrl` et par appel brut si `contrl`(1) ou `contrl`(3) sont $\neq 0$.

6, n, x1, y1, . . . , xn, yn

Trace la ligne brisée entre n points g90 b157

On peut utiliser `line` ou `polyline` de façon équivalente.

7, n, x1, y1, . . . , xn, yn

Trace n marqueurs g92 b166

On peut utiliser `mark` ou `polymark` de façon équivalente.

8, x, y, S

Affiche la chaîne S en x, y g94 b172

Identique à :

```
text x,y,S
```

9, n, x1, y1, . . . , xn, yn

Remplit le polygone à n sommets g96 b164

On peut utiliser `polyfill` de façon équivalente.

Exemple

```
for x=0,500,100
  origin x,150
  vdir 9,3,50,50,0,90,100,90
next x
```

Sortie (275 ms)

10, n, a1, a2, . . . , an

voir g98

11

Doit être appelé par les fonctions −10 à −1.

12, 0, dy

Hauteur des caractères g150 b170

Identique à `t_height` *dy*. La hauteur dy est donnée en pixels (comparer à `vdi(107)`).

13, a

Angle du texte g154 b171
Identique à `t_angle` *a*.

14, c, r, v, b

Met la couleur g132 b167 La couleur a pour index $c \in [0, 15]$. Les intensités du rouge *r*, du vert *v* et bleu *b* vont de 0 à 1000.

15, t

Type de ligne g136 b155
Identique à `l_type t`.

16, e, 0

Epaisseur des lignes g140 b156
Identique à `l_width e`. L'argument 0 est obligatoire.

17, c

Couleur des lignes g134 b167
Identique à `l_color c`.

18, t

Type de marqueur g144 b165
Identique à `m_type t`.

19, 0, h

Hauteur du marqueur g146 b166
Identique à `m_height h`.

20, i

Couleur du marqueur g148 b167
Identique à `m_color i`.

21, f

Sélectionne une fonte g156 b186
Analogue à `t_font` *f*. Voir aussi `vst_font` de la bibliothèque STND.

22, i

Couleur du texte g158 b167
Identique à `t_color i`.

23, t

Type de remplissage g165 b161
Identique à `f_type t`.

24, s

Style de remplissage g167 b161
Identique à `f_style s`.

25, i

Couleur du remplissage g169 b167

Identique à `f_color i`.

26, i, f

Composition de la couleur i g228 b168

En sortie `intout`(1) à `intout`(3) donnent les intensités du rouge, vert et bleu ($f = 0$ réelles ou $f = 1$ arrondies).

28, x, y

Souris g189 g192 b181

Analogue à `setmouse x,y`. Si en sortie `contrl`(2)=1 la souris a été déplacée jusqu'au point (`ptsout`(0), `ptsout`(1)).

29, v

Voir g195–7

30, n, t

Voir g199 (n est mis dans `contrl`(3) en entrée)

31, x, y, l, m

Entrée de chaîne au clavier g202 g204 b182

Peut remplacer `input`.

32, m

Mode graphique g128 b155

Identique à `graphmode m`.

33, p, a

Mode d'interrogation g187 b181

$p = 1$ (souris), $p = 4$ (clavier), $a = 1$ (attente), $a = 2$ (test).

35

Sortie des attributs de ligne g230 b158

36

Sortie des attributs de marqueur g232 b166

37

Sortie des attributs de remplissage g233 b164

38

Sortie des attributs de texte g235 b174

39, h, v

Alignement du texte g162 b169

100, a0, a1, ..., a11

Ouvre une station de travail virtuelle g78 g296 b151

Voir `v_opnvwk` de la bibliothèque STND.

101

Ferme la station de travail virtuelle g82 b154

Voir `v_clsvwk` de la bibliothèque STND.

102, f

Informations sur la station de travail g223 b153

103, x, y, c

Remplissage g101 b164

Identique à `fill x,y,c`

104, f

Bord des formes pleines g171 b163

Identique à `boundary f`.

105, x, y

Voir g185 b168

Analogue à `point(x,y)` ou `vpoint(x,y)`.

106, t

Effets de texte g160 b171

Identique à `t_type t`.

107, n

Hauteur de la fonte g152 b188

Voir `vst_point` de la bibliothèque STND.

108, d, f

Début et fin de ligne g142 b157

Identique à :

```
l_begin d
l_end f
```

109, x, y, xp, yp, X, Y, XP, YP, m

Copie de zone g177 b183

La fonction copie le rectangle x, y, xp, yp vers X, Y, XP, YP avec l'option logique *m*. Avant l'appel il faut donner les adresses des structures MFDB par :

```
pokels contrl+14,MFDBsource,MFDBdestination
```

Voir la procédure `bitblt` de la bibliothèque STND.

110

Copie structure MFDB g183 b185

Avant l'appel il faut donner les adresses des structures MFDB par :

```
pokels contrl+14,MFDBsource,MFDBdestination
```

111, f0, f1, ..., f36

Nouvelle forme de la souris g207 b179

Voir `defmouse` pour la signification de f0, ..., 36.

Exemple

Le programme modifie la forme de la souris, puis, après appui sur une touche, remet la flèche par `defmouse`.

```
vdi 111,0,0,1,0,1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-
 1,-1,-1,-1,1440,3024,1632,1056,15996,2064,1057,2066,41
 06,4106,4106,4106,2068,1064,592,480
ift keyget
defmouse 0
```

112, g, f1, ..., fg

Nouveau style de remplissage g173 b163

Identique à `f_user g,f1,...,fg`.

113, m

Nouveau style de ligne g138 b156

Identique à `l_type m`.

114, x, y, xp, yp

Remplit l'intérieur d'un rectangle g103 b158

Identique à `pibox x,y,xp,yp`.

115, u

Voir g242

116, S

Cadre pour la chaîne S g237 b173

Voir `vqt_extent` de la bibliothèque STND.

117, m

Information sur le caractère *m* de la fonte courante g240 b173

118

Interruption : compteur g209 b189

119, m

Charge les fontes g85 b185

Voir `vst_load_fonts` de la bibliothèque STND.

120, m

Ote les fontes g87 b187

Voir `vst_unload_fonts` de la bibliothèque STND.

121, x, y, xp, yp, X, Y, XP, YP, m, c, cp

Copie un rectangle g180 b184

122, c

Montre la souris g211 b179

Il vaut mieux utiliser `showm`.

123

Cache la souris g213 b179

Il vaut mieux utiliser `hidem`.

124

Retourne la position de la souris g214 b183
En sortie `ptsout`(0) et `ptsout`(1) donnent les coordonnées, et `intout`(0) le bouton (bit 0 gauche, bit 1 droite). Voir `mouse`.

125

Interruption : bouton g216 b189

126

Interruption : mouvement de la souris g217 b189

127

Interruption : tracé de la souris g219 b189

129, x, y, xp, yp, f

Restreint l'affichage g88 b154
Identique à `clip x,y,xp,yp,f`.

130, n

Information sur la fonte numéro *n* b186
Voir `vqt_name` et `vqt_name$` de la bibliothèque STND.

131

Information sur la fonte courante b186
Voir `vqt_fontinfoy` de la bibliothèque STND.

AES

GCONTRL

GLOBAL

GINTIN

GINTOUT

ADDRIN

ADDROUT

Constantes ou Index

En tant que constantes, ce sont les adresses de tables de nb octets permettant le passage des paramètres avec l'AES. En tant que noms indicés, les indices *i* vont de 0 à max.

	nb	max
`gcontrl`	10	4
`global`	30	14
`gintin`	32	15
`gintout`	14	6
`addrin`	12	2
`addrout`	4	0

Ces tables peuvent être initialisées ou lues en utilisant les formes indicées (correspondant à des index*16 sauf pour `addrin` et `addrout` qui correspondent à des index*32).

Exemple

```
gintin(1)=13
addrin(0)=x
w=gintout(1)
```

équivaut à :

```
pokews gintin+2,13
pokels addrin,x
w=peekws(gintout+2)
```

AES

AES k {, ini} {, adj}

Commande Appel AES

AESF(...)

V_fonction Appel AES

k

entier $k \in [10, 125]$

ini

entier*16

adj

entier*32

Nombre de renseignements donnés à propos de la commande `vdi` restent valables et ne sont pas répétés ici. Si aucun argument n'est donné, la commande `aes` effectue un appel AES brut sans initialiser les tableaux. Il faut donc initialiser les tableaux `gcontrl`, `addrin` et `gintin` avant l'appel brut AES. L'appel par

la fonction `aesf`(...), utilise les mêmes arguments que la commande `aes`. En retour, la fonction renvoie la valeur de `gintout`(0). Par exemple :

```
print aesf(77)
```

écrit l'identificateur de l'application (=1).

Δ Les tableaux `gcontrl`, ... sont également utilisés par le Basic, et le bloc de paramètres, en `systab` + 22, qui contient les adresses de ces tableaux peut être modifié.

Si l'argument k est donné Basic 1000d se charge de remplir les 5 valeurs `gcontrl`(0) à `gcontrl`(4) selon le numéro de fonction k. Ensuite les $n+1$ arguments in0, in1, ..., in$n$ sont utilisés pour remplir `gintin`(0) à `gintin`(n) et les $m+1$ arguments ad0, ..., ad$m$ servent à remplir `addrin`(0) à `addrin`(m). Bien sûr n et m dépendent de la fonction k et on a $n+1 =$ `gcontrl`(1), $m+1 =$ `gcontrl`(3). L'entrée `gcontrl`(4) est initialisée égale à 0.

Table des fonctions AES

Nous donnons les valeurs de `gcontrl`(0) à `gcontrl`(3). Le numéro de la fonction AES est $k =$ `gcontrl`(0). Le nombre d'arguments ini attendus est $\nu = n+1 =$ `gcontrl`(1). Le nombre d'arguments en sortie sur `gintout` est $s =$ `gcontrl`(2). Le nombre d'arguments adj est $\mu = m+1 =$ `gcontrl`(3). Les indications g et b sont les références bibliographiques.

	application					
k	ν	s	μ	g	b	
10	0	1	0	g294	b190	appl_init
11	2	1	1	g340	b191	appl_read
12	2	1	1	g341	b191	appl_write
13	0	1	1		b192	appl_find
14	2	1	1		b193	appl_tplay
15	1	1	1		b192	appl_trecord
19	0	1	0		b193	appl_exit

	événements					
k	ν	s	μ	g	b	
20	0	1	0	g331	b194	evnt_keybd
21	3	5	0	g332	b194	evnt_button
22	5	5	0	g334	b195	evnt_mouse
23	0	1	1	g337	b195	evnt_mesag
24	2	1	0	g336	b198	evnt_timer
25	16	7	1	g338	b198	evnt_multi
26	2	1	0		b200	evnt_dclick

menus

k	ν	s	μ	g	b	
30	1	1	1	g387	b200	menu_bar
31	2	1	1	g388	b201	menu_icheck
32	2	1	1	g389	b201	menu_ienable
33	2	1	1	g391	b202	menu_tnormal
34	1	1	2	g393	b202	menu_texte
35	1	1	1	g394	b202	menu_register

objets

k	ν	s	μ	g	b	
40	2	1	1		b203	objc_add
41	1	1	1		b203	objc_delete
42	6	1	1	g359	b204	objc_draw
43	4	1	1	g361	b204	objc_find
44	1	3	1	g362	b205	objc_offset
45	2	1	1		b205	objc_order
46	4	2	1	g363	b206	objc_edit
47	8	1	1	g365	b206	objc_change

formes

k	ν	s	μ	g	b	
50	1	1	1	g375	b207	form_do
51	9	1	1	g376	b207	form_dial
52	1	1	1	g381	b208	form_alert
53	1	1	0	g383	b208	form_error
54	0	5	1	g378	b209	form_center
55	3	3	1			form_keybd
56	2	2	1			form_button

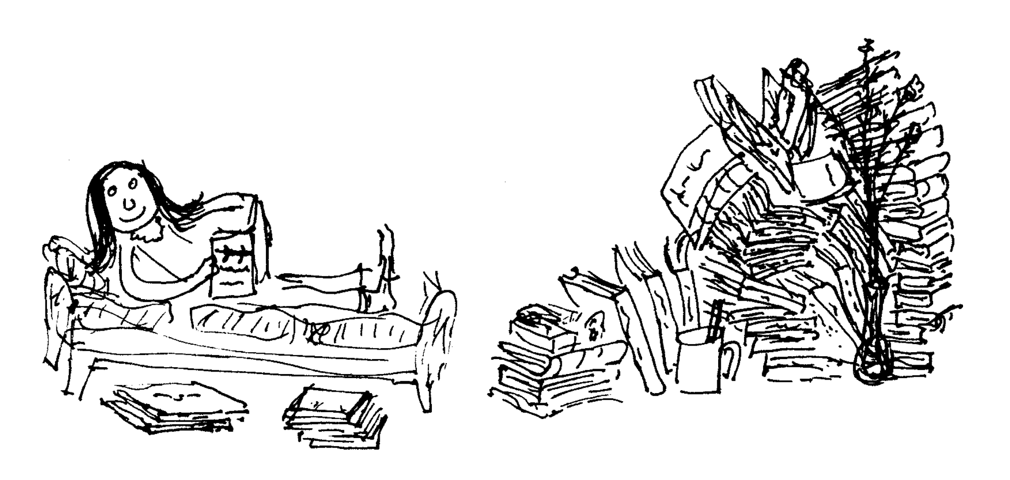
graphismes

k	ν	s	μ	g	b	
70	4	3	0	g396	b209	graf_rubberbox
71	8	3	0	g398	b210	graf_dragbox
72	6	1	0	g400	b210	graf_movebox
73	8	1	0	g402	b211	graf_growbox
74	8	1	0	g404	b211	graf_shrinkbox
75	4	1	1	g406	b211	graf_watchbox
76	3	1	1	g408	b212	graf_slidebox
77	0	5	0	g295	b212	graf_handle
78	1	1	1	g410	b213	graf_mouse
79	0	5	0	g412	b213	graf_mkstate

		divers				
k	ν	s	μ	g	b	
80	0	1	1		b225	scrp_read
81	0	1	1		b226	scrp_write
90	0	2	2		b225	fsel_input
91	0	2	3			fsel_exinput (TOS $\geq$ 1.4)
		fenêtres				
k	ν	s	μ	g	b	
100	5	1	0	g309	b214	wind_create
101	5	1	0	g311	b215	wind_open
102	1	1	0	g312	b219	wind_close
103	1	1	0	g313	b219	wind_delete
104	2	5	0	g302	b215	wind_get
105	6	1	0	g314	b216	wind_set
106	2	1	0	g317	b217	wind_find
107	1	1	0	g318	b218	wind_update
108	6	5	0	g320	b218	wind_calc
109	0	1	0			wind_new (TOS $\geq$ 1.4)
		ressources				
k	ν	s	μ	g	b	
110	0	1	1	g370	b220	rsrc_load
111	0	1	0	g371	b221	rsrc_free
112	2	1	0	g372	b220	rsrc_gaddr
113	2	1	1	g374	b221	rsrc_saddr
114	1	1	1		b221	rsrc_obfix
		communications				
k	ν	s	μ	g	b	
120	0	1	2		b222	shel_read
121	3	1	2		b222	shel_write
122	1	1	1		b223	shel_get
123	1	1	1		b224	shel_put
124	0	1	1		b223	shel_find
125	0	1	2		b223	shel_envrn

20

Bibliothèque STND



Le noyau de commandes internes du Basic 1000d gère le GDOS et les fenêtres presque exclusivement par des appels systèmes (`vdi` et `aes`). Dans divers langages, ces appels sont effectués par des instructions spécialisées. Par exemple, la fonction qui dans divers Basics ou C, ou leurs bibliothèques, est disponible sous le nom VST_UNLOAD_FONTS s'appelle en Basic 1000d par `vdi(120)`. Cependant, à l'aide d'une bibliothèque, il est également très facile de faire accepter ce nom au Basic 1000d. Il suffit d'y placer la fonction :

```
VST_UNLOAD_FONTS:function
                    value=vdif(§120,0)
                    return
```

Pour adapter un programme écrit dans un autre langage au Basic 1000d, au lieu d'effectuer une traduction complète il peut être plus simple d'utiliser une bibliothèque qui jouera le rôle d'un émulateur, au moins partiellement. Par exemple, si `vtab lg` fixe la ligne du curseur dans ce langage, au lieu de traduire `vtab` en `cursl`, on peut placer la procédure `vtab` suivante dans la bibliothèque :

```
vtab:cursl @1
     return
```

Ce chapitre décrit la bibliothèque STND, qui fournit des programmes qui sont analogues (mêmes rôles et mêmes noms) à des instructions d'autres langages. On y trouvera des exemples de gestion du GDOS, des menus déroulants et des fenêtres. De plus nous donnons des indications sur l'adaptation de programmes écrits en d'autres Basics.

On peut appeler les procédures de STND avec une base ≥ 10, et écrire leurs noms en majuscules ou minuscules (l'option `nodistingo` est valide). Pour éviter les conflits de noms, il est conseillé de ne pas utiliser les noms commençant par `g_`. La bibliothèque STND place l'origine graphique `originx`, `originy` au coin supérieur gauche de la fenêtre active. Au début d'un programme, `originy` $\neq 0$ place l'origine sous la barre des menus.

B_INIT

La procédure B_INIT, qui est automatiquement appelée par RUN ou DEBUG, initialise des variables pour la gestion des fenêtres. Elle donne la totalité de l'écran au programme. Lors du retour à l'éditeur il se peut que le damier efface les sorties. C'est justement le cas pour une grande partie des exemples proposés. Pour examiner les résultats, le plus simple est de rajouter en fin du programme :

```
ift keyget
```

ou :

```
message "Fin du|programme"
```

qui attendent l'appui sur une touche (la deuxième façon cache un bout de l'écran). On peut au lieu de cela rajouter en tête du programme :

```
cursl 4
```

qui fait partir l'impression de la ligne 4.

B_END

La procédure B_END, qui est automatiquement appelée à la fin du RUN, ferme éventuellement les fenêtres.

Menu B_USER

Le programme B_USER est un exemple de programmation des menus déroulants. On l'étudiera pour voir comment on peut retracer l'écran après les déplacements et fermetures d'accessoires. Les programmes suivants peuvent s'appeler dans ces menus.

FORMATE

Procédure Formate les disquettes
La procédure `formate` offre un choix de formats.

CALENDRIER

Procédure Calendrier perpétuel
La procédure `calendrier` utilise les calendriers julien et grégorien.

IMPORTER

Procédure Traduction
La procédure `importer` permet de traduire des programmes venant d'autres Basics en Basic 1000d. Elle est décrite plus en détail dans la section suivante.

LIBRARYP

Procédure Affiche le nom de la bibliothèque
La procédure `libraryp` est appelée lors du retour à l'éditeur du menu B_USER. Elle affiche la valeur des deux fonctions suivantes `library` et `libraryv`.

LIBRARY

C_fonction Nom de la bibliothèque

LIBRARYV

V_fonction Numéro de version de la bibliothèque

Traduction de programmes

Cette section donne des indications sur la traduction en Basic 1000d à partir d'un autre dialecte Basic.

Importer

La procédure IMPORTER, accessible par le menu B_USER, effectue des modifications sur un fichier contenant une source Basic en ASCII, pour préparer l'adaptation au Basic 1000d. Evidemment, suivant le Basic d'origine, il vous faudra modifier ce programme, qui est une simple ébauche de traducteur. Voici quelques exemples des transformations effectuées :

octets 13, 10	octet 0 (Fichier Z)
!	' (commentaires)
&H	$ (nombres hexadécimaux)
&X	% (nombres binaires)
`downto`	, (séparateur)
`end`	`stop`
`input#`	`input #` (rajoute un espace)
`print#`	`print #` (idem)

Numéros de ligne

Si le Basic à traduire utilise des numéros de lignes, il faut les remplacer par des labels. Par exemple, on remplacera :

```
        goto 1270
        ...
        1270 x=3
```

par :

```
        goto lab_1270
        ...
 lab_1270:x=3
```

Procédures

L'indication des procédures en Basic 1000d est identique à celle des sous-programmes dans de nombreux Basics anciens. Par contre si le début d'une procédure est indiqué comme dans :

```
        procedure Exemple(A,B)
```

il faut traduire en :

```
  Exemple:local datav @1,@2f var A,B
```

ou :

```
  Exemple:Procedure(A,B)
```

L'appel des procédures est en général indiqué par un mot clef (`gosub`, `@`, etc.) que l'on supprimera en Basic 1000d.

Fonctions

Dans les Basics qui acceptent des fonctions sous forme de programmes, la valeur est en général renvoyée par :

```
        return expr
```

que l'on modifiera en :

```
        value=expr
        return
```

Les instructions `deffn` doivent être transformées en fonctions et déplacées en dehors du programme. Par exemple :

```
        Deffn Yerr(I%)=Sqr(Max(Y_data(I%),I%))
        Deffn Ds_max$=Str$(N_set%)+" série(s) de données"
```

peut être transformé en :

```
    Yerr:function(index i)
        value=Sqr(Max(Y_data(i),i))
        return
 Ds_max$:function$
        value=Str$(N_set%)&" série(s) de données"
        return
```

Noter que la forme suivante, également possible, dans laquelle `@1` est calculé 2 fois, est désavantageuse si `@1` est une expression compliquée.

```
    Yerr:function
        value=Sqr(Max(Y_data(@1),@1))
        return
```

Autres difficultés

La concaténation des chaînes s'indique par `&` (Basic 1000d) et non par `+` comme dans certains Basics. Les instructions à modifier se trouvent en recherchant les suites de 2 caractères `$+` et `"+` ainsi que ces caractères séparés de caractères quelconque.

La donnée des nombres en octal n'est pas prévue en Basic 1000d (sauf en base 8). Les opérateurs logiques en Basic 1000d ont une syntaxe inspirée du langage ADA, qui diffère de celle généralement adoptée par les autres Basics. L'opérateur `not` a une précédence plus élevée que `xor`, `or` et `and`, il faut veiller à l'homogénéité (pas de mélange de `and` et `or`) et il ne peut y avoir plusieurs `imp` à la suite. Ainsi (de nombreux Basics) :

```
        not a and b
        c and d imp e imp f
```

doivent être récrites (Basic 1000d) :

```
        not(a and b)
        ((c and d)imp e)imp f
```

Mots clefs du Basic 1000d

Il faut changer les noms de variables qui sont des mots clefs. De plus, chaque nom ne peut être utilisé que pour un seul type d'objet en Basic 1000d, à la différence d'autres Basics. Les mots clefs du Basic 1000d peuvent désigner des commandes ou fonctions différentes dans d'autres Basics. Il faudra donc aussi les modifier. Voici deux mots clefs qui ont toujours des significations différentes.

CONT

En Basic 1000d, `cont` renvoie le contenu d'un polynôme. La commande CONT (reprendre l'exécution) des autres Basics s'obtient par la case DEBUG+ puis I/J Run.

LOCAL

Cette commande joue un rôle analogue dans divers Basics, mais la syntaxe étant différente, il est nécessaire de la récrire.

Commandes absentes du Basic 1000d

Comme indiqué dans l'introduction, elles peuvent en général être émulées et STND en donne de nombreux exemples.

Impressions et Graphismes

AT(x, y)

C_fonction Localise le curseur

x, y

entier

La C_fonction `at` déplace le curseur colonne x et ligne y. Elle renvoie une chaîne vide. On peut donc l'utiliser dans `print` en la faisant suivre de ";" ou "&", mais pas de ",".

Exemple

```
print at(1,10);"a";at(0,10);"b"
```

Sortie ligne 10 (20 ms)

```
ba
```

CRSCOL

CRSLIN

V_fonctions Position du curseur

La fonction `crscol`, comme `pos` et `cursc`, renvoie la colonne du curseur et la fonction `crslin`, comme `csrlin` et `cursl`, renvoie la ligne du curseur.

Exemple

```
print at(8,9);crscol;crslin
```

Sortie (25 ms)

```
 8  9
```

VSYNC

Procédure Synchronise avec le balayage écran vertical.
La procédure `vsync` équivaut à `xbios($25)`.

Exemple

Détermine la fréquence de balayage écran vertical.

```
vsync
push mtimer
for i=0,199
  vsync
next i
print using "#.#Hz";1000~*i/(mtimer-pop)
```

Sortie (2895 ms)

```
70.2~Hz
```

SETCOLOR n, r, v, b

SETCOLOR n, rvb

Procédure Fixe la couleur

n, r, v, b, rvb

entier*16

La commande `setcolor`, en posant $rvb = 256r + 16v + b$, est équivalente à :

```
color(n)=rvb
```

Exemple

En monochrome, après

```
setcolor 0,0
```

on inverse les couleurs. Pour revenir aux couleurs habituelles, on peut utiliser

```
setcolor 0,1
```

Une autre méthode pour remettre les couleurs usuelles consiste à entrer dans le débogueur, puis à en sortir par Arret (F2).

BITBLT a

BITBLT u()

Procédure Copie de bloc par la ligne A

a, u()

Les paramètres d'entrée forment un tableau de 76 octets qui est soit transmis par adresse (*a* est l'adresse du tableau), soit initialisé à partir du tableau u().

BITBLT s(), d(), p()

Procédure Copie de bloc par le VDI

s(), d()

Les tableaux s et d donnent les MFDB (memory form descriptor) des blocs source (s) et destination (d) :

s(0)	d(0)	adresse paire de départ du bloc
s(1)	d(1)	largeur en pixels du bloc
s(2)	d(2)	hauteur en pixels du bloc
s(3)	d(3)	largeur en mots du bloc
s(4)	d(4)	non utilisé
s(5)	d(5)	nombre de plans vidéo (1, 2 ou 4)

p(0), p(1), p(2), p(3)

entiers*16, coordonnées absolues du rectangle source

p(4), p(5), p(6), p(7)

entiers*16, coordonnées absolues du rectangle destination

p(8)

entier $\in [0, 15]$, mode de transfert

L'appel de la procédure `bitblt` équivaut à l'appel suivant d'une autre procédure de la bibliothèque, `bitblt_d` :

```
bitblt_d s(0), s(1), s(2), s(3), s(5), d(0), d(1), d(2)
 , d(3), d(5), p(0), p(1), p(2), p(3), p(4), p(5), p(6)
 , p(7), p(8)
```

Exemple (monochrome)

Reproduit 5 fois le haut de l'écran par `bitblt_d`.

```
print "Exemple avec écran monochrome"
ift resolution<>2 stop
for i=64,320,64
  bitblt_d xbios(2),640,400,40,1, 0,0,0,0,0, 0,0,639,63
   , 0,i,639,i+63, 3
next i
i=keyget
print /c/
```

Exemple monochrome

On réalise un défilement horizontal de l'écran. L'écran, après copie dans le tableau scr, est divisé en deux rectangles par une droite verticale. Le grand rectangle de gauche, de largeur $640 - v$ pixels, est déplacé de v pixels vers la droite. Le petit rectangle de droite, de largeur v pixels, est recopié à gauche de l'écran. En répétant ces opérations on obtient le défilement.

Les paramètres d'entrée sont placés dans un tampon de 76 octets dont on transmet l'adresse à `bitblt`. Comme ce tampon est détruit par `bitblt`, il

est nécessaire de l'initialiser avant chaque appel. La table suivante donne la description du tampon.

0	largeur du bloc en pixels
2	hauteur du bloc en pixels
4	nombre de plans vidéo destination(1, 2 ou 4)
6	premier index de couleur
8	deuxième index de couleur
10	mode combinatoire de transfert
14	décalage horizontal source
16	décalage vertical source
18	adresse du bloc source
22	double du nombre de plans source
24	nombre d'octets par ligne source
26	0 ou 2 suivant le mode
28	décalage horizontal destination
30	décalage vertical destination
32	adresse du bloc destination
36	double du nombre de plans destination
38	nombre d'octets par ligne destination
40	0 ou 2 suivant le mode
42	0 ou adresse matrice de remplissage
46	0 ou 2 pour utiliser la matrice de remplissage
48	0 ou 32 si la matrice est multicolore
50	masque de répétition du motif
52	12 mots nuls

Le mot long en 10 permet d'associer au transfert une opération logique entre source et destination. La valeur de l'octet fort de ce mot (0 à 15) fixe l'opération selon la table suivante qui indique le pixel de destination (0 ou 1) en fonction des pixels source s et d (0 ou 1).

0	0	8	`not` (s `or` d)
1	s `and` d	9	s `eqv` d
2	s `and not` d	10	`not` d
3	s	11	s `or not` d
4	(`not` s) `and` d	12	`not` s
5	d	13	s `imp` d
6	s `xor` d	14	`not` (s `and` d)
7	s `or` d	15	1

```
hidecm
index*16 aa(37),bb(37),dd(37),scr(16000)
```

```
index a,b,d,s,v
a=ptr(aa(0))
b=ptr(bb(0))
d=ptr(dd(0))
s=ptr(scr(0))
v=32
origin 0,0
for i=0,399,16
  line 0,i,640,200
next i
pokew a,640-v,400,1,1,0,$303,$303,0,0
pokel a+18,s
pokew a+22,2,80,2,v,0
pokel a+32,xbios(3)
pokew a+36,2,80,2
copy aa(0),38,1,bb(0),1
bb(0)=v
bb(7)=639-v
bb(14)=0
do
  pokecb a̲ s,peek$(xbios(3),32000)
  copy aa(0),38,1,dd(0),1
  bitblt d
  copy bb(0),38,1,dd(0),1
  bitblt d
loop
```

G_GET x0, y0, x1, y1, c$

G_PUT x2, y2, c$

SGET c$

SPUT c$

Procédures Transferts entre écran et c$

x0, y0, x1, y1

réels, coordonnées relatives d'un rectangle en pixels

c$

nomi de type char

x2, y2

réels, coordonnées relatives d'un point en pixels

La procédure `g_get` copie le rectangle dans la variable c$. Inversement `g_put` affiche ce rectangle sur l'écran, pas nécessairement au même endroit.

Exemple

Déplace horizontalement plusieurs copies d'un dessin.

```
for i=1,3
  ellipse 60,50,i*19,i*15
next i
f_type 2
f_style 9
fill 60,50
char c
g_get 0,0,120,100,c
i=0
do
  i=modr(i+8,160)
  G_put i,0,c
  G_put i+160,0,c
  G_put i+320,0,c
  G_put i+480,0,c
  G_put i-160,0,c
  ift keytest exit
loop
```

Les procédures `sget` et `sput`, inverses l'une de l'autre, concernent les 32000 octets de l'écran logique.

Exemple

Affichage de "A", stockage de l'écran dans c et effacement. Après appui sur une touche, les "A" sont remis.

```
char c
print chr$($41,900)
sget c
cls
i=keyget
sput c
i=keyget
```

SPRITE ch [, x, y]

Procédure Affiche, déplace ou efface un lutin

ch

nomi de type char

x, y

réels, coordonnées relatives

Si les coordonnées x, y sont précisées, la procédure `sprite` y affiche le lutin défini par le contenu de ch. Si le lutin était déjà affiché, il est effacé de l'ancienne position avant cet affichage. Si x et y sont omis, l'ancien lutin est effacé. Pour

le premier appel, ch doit contenir une chaîne de 74 octets, formée de 37 mots. C'est le bloc de définition du lutin, utilisé par la ligne A du système :

0	x_0
2	y_0
4	type d'affichage (0 positif, 1 négatif)
6	couleur du masque (0 ou 1)
8	couleur du lutin (0 ou 1)
10	ligne 0 du masque
12	ligne 0 du lutin
14	ligne 1 du masque
16	ligne 1 du lutin
	...
70	ligne 15 du masque
72	ligne 15 du lutin

Le vecteur $(-x_0, -y_0)$ définit une translation appliquée au lutin. Après le premier appel, ch contient en plus du lutin, le contenu de l'ancien écran sous le lutin.

Exemple

Déplace aléatoirement un lutin en forme de croix.

```
origin 25,25
f_type 2
f_style 1
pbox 0,0,266,166
char c
c=chr$(0,8)&mki$(1)
for i=0,15
  c=c&mkl$($0380 or ((i>5) and (i<9)))
next i
sprite c,0,0
while keytest=0
  hidem
  sprite c,random(250),random(150)
  pause 500
wend
```

DEFFILL i, t, s

DEFFILL i, m$

Procédure Attributs de remplissage

i, t, s

entier*16

m$

exprchaîne

La procédure `deffill` permet de définir les attributs de remplissage en une seule instruction. La première forme équivaut à :

```
f_color i
f_type t
f_style s
```

La deuxième forme définit un nouveau motif m$ et équivaut à :

```
f_color i
f_user g,f1,f2,...,fg
```

avec m$=`mki$(f1)` & ... & `mki$(fg)`

Exemple

Remplit l'écran avec un motif aléatoire.

```
origin 0,0
deffill ,mkz$(random(2^256),32)
pbox 0,0,639,400
message "Fin du|programme"
cls
```

DEFLINE t, e, d, f

Procédure Attributs de ligne

t, e, d, f

entier*16

La procédure `defline` permet de définir les attributs de ligne en une seule instruction. Elle équivaut à :

```
l_type t
l_width e
l_begin d
l_end f
```

Exemple

Tracé de divers types de lignes.

```
defline 1,1,2,1
for i=1 to 6
  defline ,2*i
  line 50,i*40,300,i*40
next i
do
  defline random(2^16),1,0,0
  line 50,280,300,280
  ift keytest exit
loop
```

DEFMARK i, t, h

Procédure Attributs de marqueurs

i, t, h

entier*16

La procédure `defmark` permet de définir les attributs de marqueurs avec une seule instruction. Elle est analogue à :

```
m_color i
m_type t
m_height h
```

Exemple

Tracé de deux marqueurs.

```
defmark 1,4,20
mark 50,150,200,150
```

Sortie (55 ms)

DEFTEXT i, t, a, h

Procédure Attributs de texte graphique

i, t, a, h

entier*16

La procédure `deftext` permet de définir les attributs avec une seule instruction, de façon analogue à :

```
t_color i
t_type t
t_angle a
t_height h
```

Exemple

Ecriture vers le haut.

```
deftext ,%1001,900,32
text 250,250,"deftext"
```

Sortie (165 ms)

GDOS

Pour pouvoir utiliser les sous-programmes décrits dans cette section le programme GDOS doit être résident.

V_OPNWK(a0, a1, . . . , a10)

V_OPNVWK(a0, a1, ..., a10)

V_fonctions Ouverture d'une station de travail

a0

Numéro du périphérique (défini dans le fichier ASSIGN.SYS du GDOS)

a_0	
1 à 10	Moniteurs
11 à 20	Tables traçantes
21 à 30	Imprimantes
31 à 40	Fichiers (metafile)
41 à 50	Plaques photographiques
51 à 60	Tablettes graphiques

a1 à a9

Ces données initialisent ce qui correspond, dans la station de travail, aux variables d'état suivantes du Basic :

a_1	`l_type`	type de ligne
a_2	`l_color`	couleur de ligne
a_3	`m_type`	type de marqueur
a_4	`m_color`	couleur de marqueur
a_5	`t_font`	fonte
a_6	`t_color`	couleur du texte
a_7	`f_type`	type du style de remplissage
a_8	`f_style`	index du style de remplissage
a_9	`f_color`	couleur de remplissage

a10

Indique le système de coordonnées (0 Normalisé, 2 en pixels)
Ces systèmes sont aussi connus sous les noms NDC (normalised device coordinates) et RC (raster coordinates).
La fonction `v_opnwk` ouvre une station de travail et renvoie l'identificateur de cette station. Elle utilise `vdi(1)`. La fonction `v_opnvwk`, qui utilise `vdi(100)`, doit être utilisée au lieu de `v_opnwk` pour les périphériques écran. On peut omettre des arguments. Les arguments a_i ($i \in [1,9]$) sont alors initialisés à 1 et a_{10} à 2 par défaut. Remarquer, dans le code de ces fonctions, que l'identificateur de l'application, `peekw`(`systab`-16) (=1 normalement), doit être mis dans `contrl`(6) avant l'appel `vdi`. En sortie les tableaux `intout` et `ptsout` contiennent des informations sur le périphérique. Par exemple les valeurs x=`intout`(0) et y=`intout`(1) donnent sa largeur et hauteur en pixels.

V_CLSWK

V_CLSVWK

V_fonctions Fermeture de la station de travail courante

Les fonctions `v_clswk` et `v_clsvwk` ferment la station de travail courante, et remettent la station de travail virtuelle du Basic. Elles sont équivalentes aux fonctions `vdi(2)` et `vdi(101)`.

V_CLRWK

V_fonction Efface le tampon de sortie

La function `v_clrwk`, qui équivaut à `vdi(3)`, vide le tampon de sortie de la station de travail.

V_UPDWK

V_fonction Transmet le tampon de sortie

La function `v_updwk`, qui appelle `vdi(4)`, provoque l'impression (si le périphérique est une imprimante) du tampon de sortie.

VST_LOAD_FONTS(m)

VST_UNLOAD_FONTS(m)

V_fonctions Charge ou efface les fontes

m

donner $m = 0$

La fonction `vst_load_fonts` charge les fontes spécifiées dans ASSIGN.SYS pour le périphérique courant. L'argument d'entrée m est réservé pour une gestion intelligente des fontes dans les versions futures du VDI. Avant d'appeler cette fonction, il faut restituer suffisamment de mémoire au système par `himem`. La fonction `vst_unload_fonts` efface ces fontes (libère la mémoire). Ces fonctions sont équivalentes à `vdi(119) et vdi(120)`. En principe, `vst_load_fonts` et `vdi` 119 renvoient le nombre de fontes chargées, mais en pratique on obtient des valeurs inutilisables (avec GDOS release 1.1).

VST_FONT(f)

V_fonction Sélectionne la fonte f

f

entier*16, identificateur de fonte

La fonction `vst_font` est l'analogue de la variable d'état `t_font`. En retour on obtient l'identificateur de la fonte effectivement retenue, qui est 1 (fonte système) si la fonte f est absente.

VST_POINT(n)

V_fonction Spécifie la hauteur de la fonte

n

entier*16

La fonction `vst_point` équivaut à `vdi(107)`. La valeur n est donnée en points (1 in = 2.54 cm = 72.27 pt), à la différence de la commande `vdi(12)`, ou de

`t_height` où la hauteur est donnée en pixels. En retour, la fonction renvoie la valeur effectivement retenue (en pt).

VQT_NAME(n)

V_fonction Identificateur

VQT_NAME$(n)

C_fonction Nom

n

entier*16 numéro de fonte

Ces fonctions (qui appellent `vdi(130)`) renvoient des informations sur la fonte numéro n. Le numéro $n = 1, 2, \ldots$ de la fonte est le numéro d'ordre dans la liste des fontes associées au périphérique courant dans le fichier ASSIGN.SYS. La fonction `vqt_name` renvoie l'identificateur de la fonte qui est un entier*16 arbitraire choisi par le dessinateur de la fonte. La fonction `vqt_name$` renvoie le nom de la fonte.

VQT_EXTENT(S, x1, y1, x2, y2, x3, y3, x4, y4)

Procédure Extension du texte

S

exprchaîne

x1, . . . , y4

nomi de variables ou index

La procédure `vqt_extent`, analogue à `vdi(116)`, met dans les variables ou index x1, . . . , y4 les coordonnées (à une translation près) d'un rectangle qui encadre S.

VQT_FONTINFOY(y1, y2, y3, y4, y5)

Procedure Informations sur la fonte courante

y1, . . . , y5

nomi de variables ou index

La procédure `vqt_fontinfoy` appelle `vdi(131)` et met dans y1, . . . , y5 les positions des lignes horizontales (inférieure, descendante, demi-ligne, ascendante et supérieure) de la fonte.

Exemple d'utilisation du GDOS

Cet exemple nécessite que le driver d'imprimante (e.g. FX80.SYS) correspondant au périphérique 21 soit disponible. La présence du GDOS est vérifiée par `gdos?`, puis de la place mémoire pour le driver et les fontes est libérée au début du programme.

Une station de travail est ouverte par la fonction `v_opnwk`, appelée seulement avec l'argument $a_0 = 21$ qui correspond à une imprimante. Les commandes graphiques du Basic qui appellent le VDI (e.g. `text` et `line`) concernent normalement la station de travail virtuelle ouverte par le Basic. Dans l'exemple, après `v_h h`, ces commandes graphiques agissent sur la sortie imprimante, et non

plus sur l'écran. Par contre `print`, qui est indépendant du VDI, continue à écrire sur l'écran.

Les valeurs xmax et ymax donnent les dimensions (ici en pixels) de la page imprimante. Le programme charge des fontes et utilise le premier jeu avec une hauteur de 10 pixels et le deuxième jeu de fontes avec une hauteur de 20 points. Il encadre le nom de la fonte, et trace un schéma montrant les lignes horizontales de la deuxième fonte. La fonction `v_updwk` effectue vraiment la sortie imprimante. En fin de programme, les fontes sont effacées de la mémoire, et la station de travail est fermée.

```
if gdos?=0
  message "GDOS non résident"
  stop
endif
himem max
himem himem-$30000
h=V_OPNWK(21)
x=intout(0)
y=intout(1)
print "xmax=";x;"  ymax=";y
v_h h
ift VST_LOAD_FONTS(0)
i=VQT_NAME(2)
t_font i
c$=VQT_NAME$(2)
print if;c$
p=20
q=VST_POINT(p)
print "Hauteur";q;" pt  ";t_height;" pixels"
c$=c$
VQT_EXTENT(c$,x1,y1,x2,y2,x3,y3,x4,y4)
x0=0
origin x0,100
text 0,0,c$
box x1,-y1,x3,-y3
c$="(Aboj)"
origin x0+x3+20,100
text 0,0,c$
VQT_FONTINFOY(y1,y2,y3,y4,y5)
x3=x3+100
line 0,y2,x3,y2
line 0,0,x3,0
line 0,-y3,x3,-y3
line 0,-y4,x3,-y4
t_font 1
```

```
        t_height 10
        x3=x3+5
        text x3,y2+5,"ligne descendante"
        text x3,0,"ligne de base"
        text x3,-y3+4,"demi-ligne"
        text x3,-y4,"ligne ascendante"
        ift V_UPDWK
        ift VST_UNLOAD_FONTS(0)
        ift V_CLSWK
        stop
```

Sortie

```
xmax=  959  ymax=  1487
  2 Swiss
Hauteur  20 pt    41 pixels
```

Fenêtres

OPENW n [, x, y]

Procédure Ouvre une fenêtre

CLEARW n

Procédure Vide la fenêtre *n*

CLOSEW n

Procédure Ferme la fenêtre *n*

FULLW n

Procédure Ouvre entièrement la fenêtre *n*

INFOW n, ch

Procédure Fixe la ligne d'information

TITLEW n, ch

Procédure Fixe le titre

WINDTAB

V_fonction Adresse d'une table

x, y

entiers, coordonnées absolues du point de contact

ch

exprchaîne

n

entier $n \in [0, 4]$

La procédure `openw` permet d'ouvrir 4 fenêtres jointives pour $n = 1$, 2, 3 ou 4, le point de contact des 4 fenêtres étant (x, y). Si $n = 0$, l'écran sauf la barre des menus forme l'écran (x, y origine des coordonnées graphiques). Les sous-programmes fournis offrent une gestion simple, mais limitée des fenêtres. La procédure `clearw` efface la fenêtre, et `fullw` ouvre complètement une fenêtre. Les procédures `titlew` et `infow` permettent de donner le titre et la ligne d'information. La fermeture des fenêtres est assurée par `closew` et également par retour à l'éditeur (examiner B_END). La table `windtab` permet l'accès aux paramètres des fenêtres.

Exemple

```
        Titlew 1,"Ouvrir la fenêtre"
        On Menu Message gosub msg
        pokew windtab+2,6
        Openw 1
        Clearw 1
        Do
          On Menu
        Loop
    msg:if menu(1)=22
          Closew 1
          cls
          stop
        endif
        if menu(1)=23
          hidecm
          fullw 1
        endif
        return
```

Divers

DIM?(T())

V_fonction Nombre d'éléments d'un tableau

T()

nom de tableau suivi de “()”

La fonction `dim?` renvoie le nombre d'éléments du tableau T.

Exemple

```
var A(100),B(15,15)
char D(9,9,9)
index*1 F(9,9,9,9)
print dim?(A());dim?(B());dim?(D());dim?(F())
```

Sortie (200 ms)

```
101  256  1000  10000
```

ADD v, x

SUB v, x

MUL v, x

DEC v

INC v

Procédures Opérations

v

nomi de type var ou index

x

expr

La procédure `add` (resp `sub`) ajoute (resp retranche) x à v, et `mul` multiplie v par x. Pour les variables (mais pas les index), ces procédures équivalent aux commandes `vadd`, `vsub` et `vmul` du Basic 1000d. Noter qu'il est impossible de définir une procédure analogue pour la division nomée `div` car `div` est déjà un mot clef.

La procédure `inc` (resp `dec`) incrémente (resp décrémente) v d'une unité.

Exemple

```
v=3
add v,-2
sub v,10-5
mul v,2+2
inc v
dec v
print v
```

Sortie (45 ms)

```
-16
```

ROUND(x [,n])

Fonction Valeur arrondie

x

réel

n

entier*16 ($n = 0$ par défaut)

La fonction `round` arrondit le nombre x à n chiffres après le point décimal.

Exemple

La valeur arrondie est un nombre exact.

```
x=exp(1)
formatx 10
print x;round(x,3)
```

Sortie (165 ms)

```
0.2718281828~ E+1  2.718000000
```

FALSE

TRUE

V_fonctions à valeur constantes

La fonction `true` (resp `false`) renvoie la valeur -1 (resp 0).

Exemple

```
print true;false
```

Sortie (20 ms)

```
-1  0
```

ARRAYFILL T(), x

Procédure Initialisation d'un tableau

T()

nom du tableau, suivi de "()"

x

expr, exprchaîne ou entier suivant le type du tableau

La procédure `arrayfill` remplit tout le tableau T avec la valeur x. Pour les index, seules les tailles *8, *16 et *32 sont acceptées.

Exemple

```
dim V(10,10),T$(10)
arrayfill T$(),"abc"
arrayfill V(),2
x=random(11)
y=random(11)
print "V(";x;",";y;")=";V(x,y)
print "T$(";x;")=";T$(x)
```

Sortie (165 ms)

```
V(  0,  4)=  2
T$(  0)=abc
```

TRUNC(x)

FRAC(x)

V_fonctions Parties entière et fractionnaire

x

réel

La fonction `trunc` est synonyme de `fix` et renvoie la partie de x avant la virgule. La fonction `frac` renvoie la partie de x après la virgule.

Exemple

```
print trunc(1.6);frac(1.6)
```

Sortie (40 ms)

```
 1  3/5
```

VAL?(ch)

V_fonction Décode une expression

ch

exprchaîne

La fonction `val?` renvoie le nombre d'octets de l'exprchaîne *ch* qui peuvent être décodés comme une expr. Des chaînes comme `v^2`, `li` (`v` de type var, `li` de type lit) sont acceptées.

Exemple

```
print val?("27 fev 1989")
```

Sortie (20 ms)

```
 3
```

SOUND canal, volume, note, octave, durée

SOUND canal, volume, #période, durée

Procédure Génération de sons

canal

vaut 1, 2 ou 3

volume

entier de 0 à 15

note

entier de 0 à 25 (1 DO, 2 DO# ,. . . , 12 SI, . . .)

octave

entier de 1 à 8 (Le LA 440 Hz se trouve dans l'octave 4)

période

entier*16

On peut donner la période (en unités de 8×10^{-6} s) au lieu de note et octave.

durée

entier, temps d'attente (en unités de 20 ms)

WAVE canal, enveloppe, forme, période, durée

Commande Forme des sons

canal

entier*16

Ajouter certains des nombres suivants pour valider les canaux et le bruit :

1	canal 1
2	canal 2
4	canal 3
8	bruit canal 1
16	bruit canal 2
32	bruit canal 3
$256k$	$k \in [0, 31]$ est la période du bruit

enveloppe

entier*16

Ajouter certains des nombres suivants pour valider l'enveloppe :

1	canal 1
2	canal 2
4	canal 3

forme

entier $\in [0, 15]$, forme de l'enveloppe

période

entier $\in [0, 65535]$, période de l'enveloppe

durée

entier*16, temps d'attente (en unités de 20 ms)

Pour supprimer le son : `wave` 0, 0. Remarquer qu'il est possible d'omettre les paramètres à la fin des procédures `sound` et `wave`. La durée est alors 0, et les autres paramètres sont inchangés.

Exemple

```
for i=0,25
  sound 1,10,i,4,5
next i
sound 1,10,#284,10
sound 1,12,10,4,20
sound 2,12,13,4,20
sound 3,12,17,4,20
```

```
wave 7,7,0,65535,200
wave 0,0
```

Sortie (5345 ms)

BMOVE s, d, l

Procédure Copie mémoire

s, d, l

entier*32

La procédure `bmove` déplace les *l* octets en *s* vers l'adresse *d*. Noter que la commande Copy du menu TOOLS permet aussi d'exécuter cette commande à partir de l'éditeur.

Exemple

La procédure `bmove` est utilisée dans la mémoire écran.

```
print /c/conc$(i=0,255 of chrp$(i))
bmove xbios(2),xbios(2)+16000,16000
```

Sortie (865 ms)

FILES path

Procédure Ecrit le répertoire

path

exprchaîne donnant le nom d'un répertoire

La procédure `files` équivaut à `print files$`(path, −1).

FILESELECT path, f, ch

Procédure Sélection d'un fichier

path

exprchaîne donnant le nom d'un répertoire

f

nom de fichier par défaut

ch

variable de type char

En sortie de la procédure `fileselect` ch est remplie avec le nom sélectionné (vide si annulation).

VOID x

Procédure Evalue l'expr x

x

expr

La procédure `void` évalue l'expr *x*, mais le résultat est perdu.

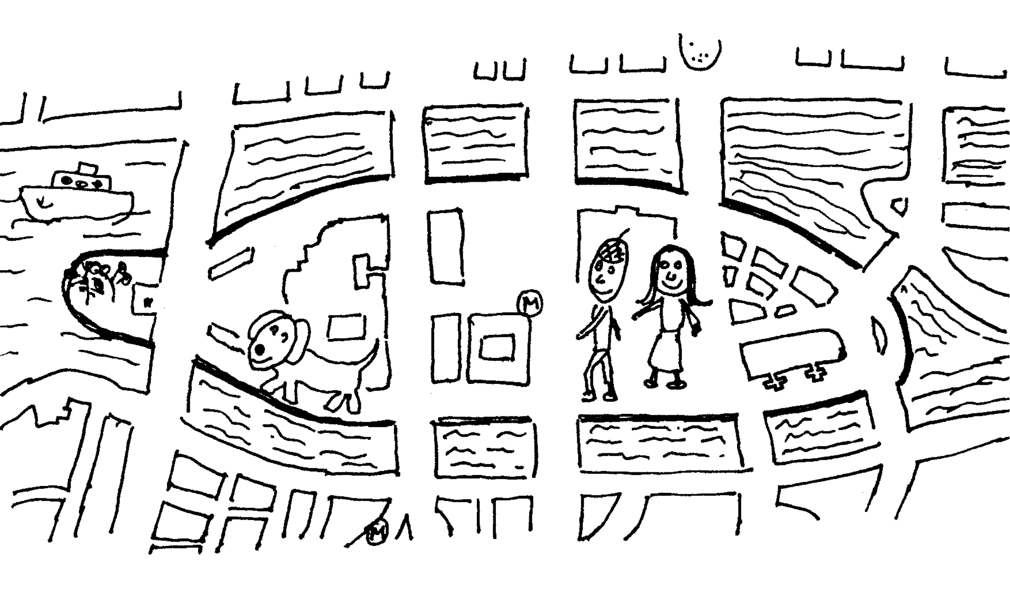
Instructions rebaptisées

La table suivante donne des sous-programmes équivalents à des fonctions et commandes du Basic 1000d.

fonctions	Basic 1000d
`vdibase`	`himem`
`peek`(x)	`peekb`(x)
`dpeek`(x)	`peekw`(x)
`lpeek`(x)	`peekls`(x)
procédures	Basic 1000d
`edit`	`stop`
`resume` *label*	`goto` *label*
`swap` x, y	`exg` x, y
`vdisys` n	`vdi` n
`gemsys` n	`aes` n
`poke` x, y	`pokeb` x, y
`spoke` x, y	`pokeb` x, y
`dpoke` x, y	`pokew` x, y
`sdpoke` x, y	`pokew` x, y
`lpoke` x, y	`pokel` x, y
`slpoke` x, y	`pokel` x, y

21

Cœur du Basic 1000d



Ce chapitre est destiné aux (futurs) experts en Basic 1000d. De nombreux renseignements sur le code du Basic sont donnés ici. Les fonctions décrites accèdent à l'intérieur du Basic.

Codages mémoire

Codage des nombres exacts

Un entier positif $m < 2^{13}$ est codé sur un mot, avec $\text{bit}_{14} = 1$. Un entier $m \geq 2^{13}, m \in [65536^{k-1}, 65536^k[$ est codé sur $k+1$ mots. Le premier mot contient $2k$ (qui doit être inférieur à 2^{13}) et les k mots suivants contiennent m, en commençant par le mot le plus significatif. Une fraction m/n avec $n \neq 1$ est codée par les deux entiers mis à la suite m puis n, avec $\text{bit}_{13} = 1$ dans le premier mot. Dans le premier mot également, $\text{bit}_{15} = 1$ indique un nombre négatif.

nombre	codage
0	$4000
1/2	$6001, $4002
-2^{15}	$8002, $8000

Codage des nombres flottants

Un nombre flottant réel est codé par un mot $k \in [-2^{15}, 2^{15}[$ suivi d'un entier exact A. La valeur du nombre est $2^k A$. La longueur de l'entier A correspond à la précision choisie. Un nombre complexe flottant est codé par deux nombres flottants réels juxtaposés, la partie réelle suivie de la partie imaginaire. Dans une variable contenant un nombre flottant, le nombre est précédé d'un mot $s = -1$ (réel) ou $s = -2$ (complexe) qui caractérise le type de valeur de la variable.

Codage d'un polynôme

Nous examinons le codage du polynôme :

$$\sum_{i=0}^{m-1} c_i x_1^{a_{i,1}} \times x_2^{a_{i,2}} \times \cdots \times x_n^{a_{i,n}}$$

Les littéraux sont ordonnés suivant leurs numéros par $X_1 < X_2 < \cdots < X_n$ et les monômes sont ordonnés dans l'ordre lexicographique décroissant des exposants. Le codage et l'impression des polynômes, pour un ordre de définition donné des littéraux, se fait toujours de la même façon. Notons enfin que dans une variable

contenant un polynôme, le codage du polynôme est précédé d'un mot nul $s = 0$, pour le distinguer des formes produit de polynômes et nombre flottant.

longueur		
1 mot	n	nombre des littéraux x_j (j de 1 à n)
n mots	X_j	pour j de 1 à n, numéro interne du littéral x_j
1 mot	$m-1$	nombre de monômes diminué de 1
	pour chaque monôme (i de 0 à $m-1$) :	
n mots	$a_{i,j}$	pour j de 1 à n, exposants
variable	c_i	nombre rationnel exact

Nous avions indiqué que les exposants des littéraux devaient être dans $[0, 2^{15}[$. En réalité, comme les exposants sont codés par un mot non signé, des valeurs dans $[2^{15}, 2^{16}[$ sont aussi possibles. Les fonctions qui utilisent uniquement des formes développées, y compris de façon interne, fonctionnent correctement avec ces valeurs. Par contre, dès que des formes factorisées sont utilisées, les résultats peuvent être faux. Ainsi la factorisation d'un polynôme contenant des exposants $> 2^{15}$ est incorrecte (car les exposants des formes factorisées sont limités à 2^{15}). Il est impossible de créer directement un tel polynôme, par exemple par `w=X^(2^15)` parce que l'exponentielle exacte exige un exposant dans $[-2^{15}, 2^{15}[$.

Codage d'un produit de polynômes

Le produit de polynômes

$$P_1 \prod_{i=2}^{s} P_i^{k_i}$$

est codé par :

longueur		
1 mot	s	nombre de facteurs ($0 < s < 65534$)
variable	P_1	nombre rationnel exact
	pour chaque facteur i de 2 à s (s'il en existe) :	
mot long		nombre d'octets du codage de P_i et k_i
variable		codage du polynôme P_i
1 mot	k_i	exposant

De plus les P_i ($i > 1$) sont ordonnés dans l'ordre de longueur croissante, puis dans l'ordre lexicographique pour les mots du codage. Le premier mot s permet aussi de distinguer la forme produit des formes polynôme et flottantes.

Accès au cœur du Basic 1000d

PTRPTR(n)

V_fonction Adresse mémoire pointant le type de n

n

nom

t	
0–C	V_fonction
D–F	commande
10–1F	lit
20–2F	index
30–3F	var
40–4F	char
50	label/proc
51	label effacé
52	label de V_fonction
54	label de C_fonction
60–6C	C_fonction
70	constante
71	`then`
72	`of`, `datav`, ...
73	`value`
74	variable d'état
75	`xor`, `or`, `and`
76–79	variable d'état
78	C_fonction et commande
79	V_fonction et commande

Si n est un nom de type inconnu, `ptrptr` renvoie 0. Si n est un mot clef ou un nom défini par l'utilisateur (ne pas écrire les indices pour un nom indicé), `ptrptr` renvoie une adresse pointant dans la description de n. Devant cette adresse on trouve les caractères du nom et sa valeur `distingo`/`nodistingo`. Encore devant on trouve un mot long qui pointe vers le nom suivant. Les noms sont organisés en environ 1000 listes liées. Le décodage d'un nom consiste à déterminer la liste du nom (qui est calculée par une opération sur les caractères du nom), puis à localiser le nom dans sa propre liste. La localisation d'un nom prend un temps négligeable devant les calculs mathématiques, parce que chaque liste est courte, et le fait que Basic 1000d (à la différence de la majorité des interprètes Basic qui

codent les noms) travaille directement sur le code ASCII n'est pas pénalisé par des temps de décodage longs.

A l'adresse `ptrptr`(n) on trouve le type t du nom n (un octet). Dans le cas des V_ ou C_fonctions, $b =$ `modr`(t, 16) indique le type du premier argument attendu. Dans le cas d'une variable, d'un littéral ou index, b est le nombre d'indices.

Indiquons maintenant, ce que l'on trouve après l'octet t, suivant le type du nom, éventuellement en sautant un octet pour atteindre une adresse paire.

label

On trouve l'adresse du ":" après le label dans la source ou bibliothèque.

index

On trouve le mot s donnant le nombre de bits b de l'index :

b	1	2	4	8	16	32
s	$703	$602	$401	0	1	3

On trouve ensuite, si l'index a k indices, k mots longs donnant les dimensions, puis un mot long contenant `ptr`(n), qui est l'adresse suivante sauf pour le type accès.

lit, char, var

On trouve le numéro de n (ou de n(`max`) si n est indicé) sur 2 octets, puis les dimensions sur k mots.

La valeur de l'index n est codée à l'adresse fixe `ptr`(n) qui est le plus souvent à la suite de la description de n, mais dans le cas de la variable n, sa valeur en `ptr`(n) se trouve dans une zone dynamique. Les littéraux n'ont pas de valeur.

WORDS$

C_fonction Liste des noms

Pour chaque mot clef et chaque nom n défini avant l'appel, la fonction `words$` renvoie le nom et le type t en hexadécimal. Ces indications sont renvoyées sous forme d'un fichier formé de lignes séparées par `chr$`(13), `chr$`(10) et suivies d'une ligne donnant le nombre de noms.

La liste se conforme à l'ordre interne, qui n'est pas alphabétique. Comme expliqué plus haut (voir `ptrptr`), elle est formée d'un millier de petites sous-listes de longueur maximum Q, dont la valeur est également sortie.

Exemple

Le programme suivant imprime tous les mots clefs du Basic dans l'ordre alphabétique. Remarquer que l'on a utilisé `push$` et `pop$` dans les deux premières lignes au lieu d'un simple `C$=WORDS$` pour éviter que le nom `C$` se retrouve dans la liste. Pour faire une sortie écran, dérouter l'impression par `lprint_dev "vbs:"`.

```
PUSH$ WORDS$
```

```
C$=POP$
OPEN "I",#0,"MEM:",C$
char D(1000)
index P(1000)
X=0
WHILE NOT EOF(0)
  LINE INPUT #0,D(X)
  X=X+1
WEND
X=X-1
lprint D(X)
SORT D(0),X,1,P(0)
M=50
FOR I=0,X-1,3*M
  FOR J=I,MIN(I+M-1,X-1)
    lprint justl$(D(P(J)-1),30);
    IF J+M<X
      lprint justl$(D(P(J+M)-1),30);
      IFT J+2*M<X lprint D(P(J+2*M)-1);
    endif
    lprint
  next J
  lprint CHR$(12);
next I
```

Sortie imprimante (30 s)
[imprime la liste sur trois colonnes]

VARNUM(V)

V_fonction Numéro de la variable V

VARN(k)

CHARN(k)

Accès aux variables

V

nomi de variable (de type var ou char)

k

entier*16 $k > 0$

La fonction `varnum` renvoie un entier dans $[0, 2^{15}]$. La variable V de type var (resp char), de numéro $a =$ `varnum`(V), est aussi accessible sous le nom `varn`($2^{15}-1-a$)

(resp `charn`($2^{15}-1-a$)). `varn` donne également accès à des variables internes :

a		
1	`varn($7FFE)`	i^2+1 (après `complex` i)
2	`varn($7FFD)`	π
4	`varn($7FFB)`	log 2
6	`varn($7FF9)`	première condition (commande `cond`)
7	`varn($7FF8)`	deuxième condition

Exemple

La deuxième instruction équivaut à C=`mkx$`(1458). Elle assigne une expr à la variable de type char C. Cette valeur est ensuite relue par un `push$` suivi de `pop`.

```
char C
varn($7FFF-varnum(C))=1458
push$ C
print pop
```

Sortie (20 ms)

```
1458
```

Attention l'utilisation de k correspondant à des variables non créées est accepté dans `varn`(k) et `charn`(k) et conduira probablement à un plantage.

TYPCHR

V_fonction Adresse de la table du type des caractères

L'octet à l'adresse `typchr` + k (où $k \in [0,255]$) donne des indications sur le caractère `chr$`(k), suivant les bits 1–7 de cet octet :

bit	propriétés du caractère
1	accepté pour CLEF dans Help
2	n'est pas un séparateur
3	implicit char
4	implicit index
5	autre que `chr$`(0) et “”(fin d'instruction)
6	lettre
7	alphanumérique (pour les noms)

En modifiant cette table on modifiera le rôle du caractère. La table `typchr` usuelle est réinitialisée par `clear`.

Exemple

La modification de la table `typchr` modifie les propriétés du caractère #.

```
        pokeb typchr+$23,32
        open "o",1,"vbs:"
        print#1,"# n’est plus alphanumérique"
```

Sortie (40 ms)

```
# n’est plus alphanumérique
```

Allocation de la mémoire

Carte de la mémoire

La position en mémoire des divers segments décrits ici s’obtient en partie par la commande éditeur = ou Mem map du menu TOOLS. La zone mémoire allouée par le GEM au Basic (allocation) va de `basepage` à `himem`.

Basepage

Cette adresse est renvoyée par la fonction `basepage`. Les 256 octets de la page de base sont suivis du programme Basic 1000d (≈ 120 koctets de code 68000), puis des tables fixes du Basic (≈ 50 koctets).

Help

Fichier du programme d’aide

Bibliothèque

Codes en Basic

Source

Codes en Basic

Tampon source

Tampon de `s_src` octets, permet de modifier la source sans détruire les zones mémoires plus hautes. La reprise de l’exécution d’un programme reste alors possible.

Pile des procédures/boucles

Zone de `s_pro` octets servant, à partir du bas à stocker les appels de procédures, et à partir du haut les indications de boucle. Cette pile contient également les descriptions des éléments locaux, ainsi que les valeurs des index locaux.

Pile des exécutions

Zone de `s_xqt` octets servant à empiler les textes développés d’instructions contenant des `@` ou des commandes `xqt`.

Pile des fichiers “MEM:”

Zone pour `mem_files` fichiers virtuels.

Tampon des menus

Zone de `s_menu` octets, pour la gestion des menus.

Table `remember`

Elle comporte `s_rem` éléments. Chaque lecture d'un ancien élément augmente sa priorité, et une nouvelle entrée se fait avec la priorité maximum au détriment de l'élément le moins prioritaire qui est effacé.

Table des noms/index

Contient les descriptions des éléments globaux (index, variables, littéraux et labels), ainsi que les valeurs des index globaux.

Tampon des noms

Zone de `s_name` octets environ, servant à inscrire les noms inconnus lors d'une instruction donnée. Après l'exécution de l'instruction, la longueur de cette zone est restaurée par déplacement vers le haut de toute la mémoire au-dessus. `s_name` n'a besoin d'être modifié que si vous utilisez une instruction contenant un trop grand nombre de noms inconnus.

Adresses des variables

A chaque variable correspond un mot long donnant l'adresse de son contenu. Par numéros croissants, les variables sont classées en :

• Variables internes

• Variables conditions

`s_cond` variables contenant les conditions (commande `cond`) du programme.

• Variables fichiers “R”

`r_files` variables de type char contenant les informations sur les fichiers à accès sélectif.

• Variables `remember`

`s_rem` variables mémorisant les valeurs des fonctions.

• Variables des piles interne et utilisateur

Cela correspond à un ensemble de `s_var` variables. L'ensemble est utilisé à partir du bas, pour stocker les `value` des fonctions, les valeurs intermédiaires lors des calculs d'expressions et par certaines fonctions (comme `det` ou `elim`). C'est la pile interne. Les variables locales y sont également empilées. L'ensemble est utilisé à partir du haut, pour les commandes `push`, `push$`, les fonctions `pop` et `pop$` et les accès `stack` et `stack$`.

• Variables utilisateur

Ce sont les variables définies par le programme.

Description des variables

Cette zone s'étend jusqu'à `limit`. Elle est gérée suivant la valeur de `pack`. Le contenu d'une variable V est précédé d'un mot, en `ptr`(V)−6, indiquant

le numéro de la variable et d'un mot long, en `ptr`(V) − 4, indiquant la longueur du contenu. Lors d'un changement de valeur de V, l'ancienne description est marquée effacée (sur le numéro de V), mais n'est pas physiquement effacée, et l'adresse du contenu de V (dans la zone précédente) est placée sur la nouvelle description de V. Cependant, lorsque après cette nouvelle description (qui a été placée en haut de la zone des descriptions), l'espace restant est inférieur à `pack`, le programme effectue un nettoyage en supprimant les anciennes descriptions. Ce nettoyage, appelé ramassage des poubelles, est également effectué lors de l'appel de certaines fonctions ou commandes (comme `pack`, `ptr`, `fre`).

Le haut de cette zone est utilisé comme pile du processeur 68000.

Réserve

Cette zone, qui part de l'adresse `limit` n'est pas utilisée directement par le Basic, et est donc à la disposition du programmeur. Au chargement du Basic cette zone est de longueur nulle.

Himem

Cette zone, de longueur modifiable par la commande `himem`, n'appartient pas au Basic. Sa longueur minimale est $4000 octets, ce qui permet un fonctionnement correct de l'AES.

Ecran=`peekls`($436)

Habituellement l'écran occupe le haut de la mémoire.

Les Variables d'état de structure

La carte mémoire dépend de variables d'états, comme `s_src`, qu'il est possible de lire et de modifier. Cependant, une modification comme :

```
s_src 2000
```

provoque un réarrangement complet de la mémoire, suivi d'un `clear`. Il est donc impossible de modifier ces variables d'état dans une procédure.

S_SRC a

S_NAME a

S_XQT a

S_PRO a

S_MENU a

S_VAR k

S_COND m

R_FILES m

MEM_FILES m

S_REM m

Variables d'état

a

entier*32 pair $a \geq 512$

k

entier*16 $k \geq 100$

m

entier*16 $m \in [1, 2^{13}[$ `s_menu`, `s_var`, `s_rem`

La signification de ces variables d'état est détaillée avec la description de la carte mémoire dans la section précédente. En pratique (`s_rem` excepté), on modifiera leurs valeurs en tête d'un programme, seulement après un message du Basic ayant indiqué une valeur trop faible.

PACK [p]

Variable d'état Conditionne le nettoyage de la mémoire

p

entier*32 pair $p > 2000$

La commande `pack` sans argument force un nettoyage sans modifier la valeur de `pack`. Noter que si on donne à la variable d'état `pack` une valeur plus grande que l'espace libre `fre`, on force un réarrangement mémoire plusieurs fois par instructions. Les temps de calcul peuvent s'en ressentir très défavorablement. Il est préférable (temps de calcul) d'utiliser une valeur aussi faible que possible pour `pack`, c'est à dire de l'ordre de grandeur des variables manipulées par le programme (pour éviter l'erreur mémoire).

LIMIT adressepaire

LIMIT MAX

Variable d'état Haut de la mémoire

La variable d'état `limit` concerne le haut de la mémoire utilisée par le Basic 1000d. La zone réservée, c'est à dire la zone mémoire [`limit`, `himem`[, est inutilisée par le Basic (de façon interne) et le système.

Exemple

Le mot clef `limit` est utilisé comme commande puis comme variable. On peut poker librement dans la zone réservée, qui est ensuite supprimée par `limit max`.

```
limit $70000
pokels $70000, 1,$23456789
print peekls(limit);peekls(limit+4)
limit max
```

Sortie (35 ms)

```
  1  591751049
```

BASEPAGE

V_fonction Adresse de la page de base

HIMEM k

HIMEM MAX

Variable d'état Haut de l'allocation.

k

adresse paire

La zone mémoire de `basepage` à `himem` est allouée au Basic par le système. La zone au dessus de `himem` est donc utilisable par d'autres programmes. Par exemple :

```
himem himem-100000
```

donne 100000 octets de plus au système, qui sont pris sur l'allocation du Basic. La commande :

```
himem max
```

laisse $4000 octets au système, et donne au Basic la plus grande place possible.

La commande `himem` modifie `limit` si nécessaire. Si après avoir diminué `himem` on effectue des allocations mémoires par `gemdos($48)`, il faudra en général libérer ces zones, par `gemdos($49)`, avant de pouvoir réaugmenter `himem`.

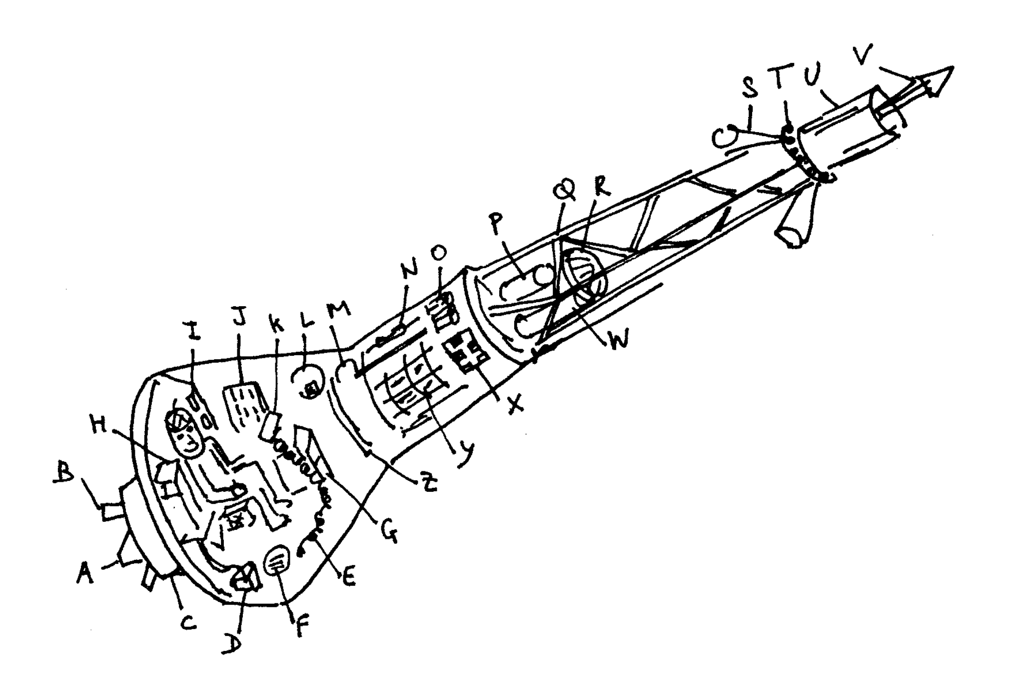
SYSTAB

V_fonction Adresse table variables internes

L'adresse `systab` permet l'accès aux variables internes du Basic suivantes (on donne les adresses en décimal par rapport à `systab`, B=octet, W=mot et L=mot long).

−16	W	identificateur de l'application (= 1 normalement)
2	5*L	contrl, intin, ptsin, intout, ptsout (bloc VDIPB)
22	6*L	gcontrl, global, gintin, gintout, addrin, addrout (bloc AESPB)
46	20*B	option `print /E/`
66	20*B	option `print /I/`
90	128*B	table de conversion pour l'impression de la source

Appendice



Solution des exercices

Apluslong

Utilisons le menu FND/CHG. Définir (case F1) S comme :

f`A`*f*

et chercher S (case F8). Cela cherche les `A` entre séparateurs, car le label `A` ne peut apparaître qu'entre séparateurs. Revenir dans FND/CHG et définir (case F2) T comme :

```
apluslong
```

et chercher (case F9) cette chaîne. Supposons qu'elle n'existe pas, on obtient le diagnostic "pas trouvé". Revenir a FND/CHG et changer (case F10).

Si maintenant vous avez un regret, et trouvez que le nouveau nom est trop long, vous pouvez annuler le changement comme suit (parce qu'il n'y avait pas de chaîne `apluslong` à l'origine). Revenir dans FND/CHG et permuter S et T (case F3), puis changer (case F10).

Nb_nom

Un nom commence par une des 52 lettres majuscules et minuscules. Les caractères suivants peuvent être aussi un chiffre ou "!#$%.?_" soit $N = 52+10+7$ possibilités. Il y a $52N^{k-1}$ noms de k caractères, et k peut aller de 1 à 32. La solution ci-dessous ne tient pas compte des mots clefs du Basic.

```
print "Le nombre de noms différents possibles en Basic
 1000d est"a;sum(k=1,32 of 52*69^(k-1))
print "Mais avec nodistingo ce nombre n'est plus que"a;
 26*(43^32-1)/42
```

Grand

Le calcul du nombre est immédiat (0.4 s), mais l'impression prend 4 minutes environ (c'est le temps de conversion de la représentation binaire en décimal).

```
print /c/"Calcul du plus grand entier de Basic 1000d"
W=4^32759*2
W=W-1+W'2^65520-1
print "Temps du calcul (ms)=";mtimer
print "Ce nombre est codé sur";mlen(W);" octets  par:"
print/h/conc$(i=0,10,2 of peekw(ptr(W)+i))&"  ..."
char C
print "Impression du nombre"
clear timer
C=justl$(W)
print "Temps d'écriture=";timer
```

```
        print "Ce nombre s'écrit avec";len(C);" chiffres"
        print "Appuyez sur une touche"
        i=keyget
        print /a/C
```

Max Detprm

Par suite des propriétés des déterminants, il suffit de considérer les matrices :

$$\begin{pmatrix} \cdot & \cdot & a \\ \cdot & \cdot & b \\ c & d & 9 \end{pmatrix}$$

où $a < b$, $c < d$ et $a < c$. On effectue une boucle sur les permutations de 1, ..., 8, en laissant 9 en place, et en ne calculant que les déterminants du type ci-dessus.

```
        index P(2,2),S(2,2)
        sup=1
        k=nextperm(9,P(0,0),0)
        while k
          ift P(0,2)>P(1,2) goto nk
          ift P(2,0)>P(2,1) goto nk
          ift P(0,2)<P(2,0) goto nk
          x=abs(det(P,2,0))
          ift x<=sup goto nk
          if prtst(x)
            sup=x
            for i=0,2
              for j=0,2
                S(i,j)=P(i,j)
            next j,i
            print sup;conc$(i=0,2 for j=0,2 of S(i,j))
          endif
       nk:k=nextperm(8,P(0,0))
        wend
        print "timer=";timer
        print "matrice"
        print conc$(i=0,2 of conc$(j=0,2 of S(i,j))a̲)
```

Déterminant

La solution suivante est beaucoup plus simple que la fonction `dete`, mais nécessite un `s_var` plus grand.

```
        print detp(A,B,C,D)
        stop
   detp:function(A(root(@0,2)-1,root(@0,2)-1))
        value=det(A,root(@0,2)-1,0)
        return
```

Boulanger

Le problème peut se résoudre à l'aide de la fonction génératrice :

$$\frac{1}{(1-x)(1-x^5)(1-x^{10})} = 1 + x + \cdots + D_n x^n + \cdots$$

où D_n est le nombre de façons de former n francs.

```
W=taylor(1/[(1-x)*(1-x^5)*(1-x^10)],50)
print " Le nombre de façons de former N francs avec des
  pièces de 1,5 et 10 francs est le coefficient de x^N
 dans"
print W
```

Fibo

Le résultat s'obtient en 3 s par de simples additions.

```
W1=0
W2=1
for I=1,999
  vadd W1,W2
  exg W1,W2
next I
W1=timer
print "Le millième nombre de la suite de Fibonacci s'éc
 rit avec"&len(justl$(W2))&" chiffres "
print "C'est ";W2
print "Ce calcul a pris";W1;" secondes"
```

Grandissime

Les entiers positifs sont codés par un polynôme en z, où z représente un grand entier de valeur v_z (on a choisi $v_z = 10^{1000}$). Le plus grand nombre possible dans cette représentation n'est plus limité que par la mémoire de votre ordinateur. Les additions, soustractions et multiplications se réduisent aux mêmes opérations sur les polynômes.

La fonction `normal`(w) récrit le polynôme de sorte que tous les coefficients soient $< v_z$ (forme normale). Pour éviter les dépassements, il faut l'appeler de temps en temps.

La fonction `pwr`(w, k) calcule w^k et renvoie une forme normale.

La C_fonction `out_c`(w) transforme le polynôme en une chaîne pour impression du nombre. Le programme est assez simple avec v_z puissance de 10.

Enfin la procédure `parfait` p, calcule et écrit le nombre parfait correspondant à p. Pour $p = 216091$, on obtient un nombre parfait de 130100 chiffres, avec un temps de calcul d'une heure.

```
init
parfait 44497
parfait 86243
parfait 132049
```

```
        parfait 216091
        stop
    init:lit z
        var vz
        index lz
        lz=1000
        vz=10^lz
        pack 500000
        return
   out_c:function$(w)
        local index i,j char c
        j=deg(w,z)
        for i=j,0
          c=justl$(coef(w,z,i))
          ift i<>j c=right$(chr$($30,lz)&c,lz)
          cadd value,c
        next i
        return
  normal:function
        value=@1
        local var a,b index i
        i=0
        repeat
          a=coef(value,z,i)
          b=a-modr(a,vz)
          ift b vadd value,b/vz*z^(i+1)-b*z^i
          i=i+1
        until deg(value,z)<i
        return
     pwr:function(a,n)
        local datav 1 var v
        while n
          ift modr(n,2) v=normal(v*a)
          n=divr(n,2)
          ift n a=normal(a*a)
        wend
        value=v
        return
 parfait:procedure(p)
        clear timer
        w=pwr(2,p-1)
        w=normal(w*(2*w-1))
        char c
        c=out_c(w)
```

```
print /a/"Pour p=";p;"  le nombre N s'écrit avec";len(c
 );" chiffres. timer=";timera;"N=";c
return
```

piR2

```
precision 100
print "La surface du cercle est";pi/49
print "Ce calcul a pris";timer;" secondes"
stop
```

Mille

```
P=1
W=1
for i=1,1000
  W=prime(W+1)
  vmul P,W
next i
print "Ce calcul a pris";timer;" secondes"
print /a/"Le produit des 1000 premiers nombres premiers
  vaut";P
stop
```

Hilbert

```
print "Le déterminant vaut";det(A,10)
print "Ce calcul a pris";timer;" secondes"
stop
A:function
value=1/(@1+@2)
return
```

11…11

Le programme suivant trouve 4 tels nombres en 3 heures.

```
k=2
do
  k=prime(k)
  a=val(chr$($31,k))
  ift prtst(a) print "Le nombre écrit avec";justr$(k,5)
   f;chr$(34);"1";chr$(34);" est premier   timer=";time
   r
  k=k+1
loop
```

Goldbach

```
g=6
g1=g/2+1
do
```

```
   p=3
a1:p=prime(p)
   ift p>g1 goto a2
   ift prtst(g-p) goto a3
   vadd p,1
   goto a1
a2:print "Conjecture de Goldbach fausse"
   print g;" n'est pas la somme de 2 nombres premiers"
   print "Publiez votre résultat"
   stop
a3:print "Conjecture de Goldbach vérifiée pour";g;"=";p;
 "  +";g-p;"   timer=";timer
   vadd g,2
   vadd g1,1
 loop
```

Parfait

Le programme suivant trouve les 10 premiers nombres parfaits en 29 secondes et les 4 suivants en 1h20.

```
p=2
do
  p=prime(p)
  Mp=2^p-1
  ift prtst(Mp) print "p=";justl$(p,6);"N=";2^(p-1)*Mp;
   " (TEMPS=";timer;")"
  p=p+1
loop
```

Bibliographie

E H Bareiss, *Sylvester Identity and Multistep Integer preserving Gaussian Elimination, Math Comp* **22** (1968) 565

L Besle, *Au Cœur de l'Atari ST* (Eyrolles 1986)

J Brillhart & M A Morrison, *A Method of Factoring and the Factorisation of* F_7, *Math Comp* **29** (1975) 183

B Buchberger, G E Collins & R Loos (éditeurs) *Computer Algebra* (Springer-Verlag 1982)

J Davenport, Y Siret & E Tournier, *Calcul Formel* (Masson 1987)

R W Gosper, *Proc Nat Acad Sci USA* **75** (1978) 40

A P Jucys & A A Bandzaitis, *Teoriia Momenta Kolichesttva Dvizheniia v Kvantovoi Mekhanike* (Vilnius 1965)

D E Knuth, *The Art of Computer Programming* (Addison-Wesley 1981), vol 2

N Koblitz, *A Course in Number Theory and Cryptography* (Springer-Verlag 1987)

H W Lenstra, Jr., *Factoring Integers with Elliptic Curves*, *Ann Math* **126** (1987) 649

J A Nelder & R Mead *Computer J* **7** (1965) 308

J M Pollard, *A Monte-Carlo Method for Factorisation*, *BIT* **15** (1975) 331

H Rademacher, *On the Partition Function* $p(n)$, *Proc London Math Soc* **43** (1937) 241

D Slowinski, *J. Recreational Mathematics* **11** (1979) 258

G Szczepanowski, *Le Livre du GEM* (Micro Application 1985)

Index

Un mot de l’index ne se trouve pas nécessairement dans les pages référencées. Par exemple, le mot scrolling, renvoie à la page décrivant `cursh`, où on parle d’impression qui défile, mais pas de scrolling. On trouvera à l’entrée “table” une liste de tables du manuel.

Table des codes ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	0 16	32	0 48	@ 64	P 80	` 96	p 112	Ç 128	É 144	á 160	ã 176	ĳ 192	ס 208	α 224	≡ 240
1	⇧ 1	1 17	! 33	1 49	A 65	Q 81	a 97	q 113	ü 129	æ 145	í 161	õ 177	Ĳ 193	ע 209	β 225	± 241
2	⇩ 2	2 18	" 34	2 50	B 66	R 82	b 98	r 114	é 130	Æ 146	ó 162	Ø 178	א 194	פ 210	Γ 226	≥ 242
3	⇨ 3	3 19	# 35	3 51	C 67	S 83	c 99	s 115	â 131	ô 147	ú 163	ø 179	ב 195	צ 211	π 227	≤ 243
4	⇦ 4	4 20	$ 36	4 52	D 68	T 84	d 100	t 116	ä 132	ö 148	ñ 164	œ 180	ג 196	ק 212	Σ 228	⌠ 244
5	5	5 21	% 37	5 53	E 69	U 85	e 101	u 117	à 133	ò 149	Ñ 165	Œ 181	ד 197	ר 213	σ 229	⌡ 245
6	6	6 22	& 38	6 54	F 70	V 86	f 102	v 118	å 134	û 150	ª 166	À 182	ה 198	ש 214	µ 230	÷ 246
7	7	7 23	' 39	7 55	G 71	W 87	g 103	w 119	ç 135	ù 151	º 167	Ã 183	ו 199	ת 215	τ 231	≈ 247
8	✓ 8	8 24	(40	8 56	H 72	X 88	h 104	x 120	ê 136	ÿ 152	¿ 168	Õ 184	ז 200	ן 216	Φ 232	° 248
9	9	9 25	) 41	9 57	I 73	Y 89	i 105	y 121	ë 137	Ö 153	⌐ 169	¨ 185	ח 201	ך 217	Θ 233	• 249
A	🔔 10	ə 26	* 42	: 58	J 74	Z 90	j 106	z 122	è 138	Ü 154	¬ 170	´ 186	ט 202	ם 218	Ω 234	· 250
B	♪ 11	ES 27	+ 43	; 59	K 75	[91	k 107	{ 123	ï 139	¢ 155	½ 171	† 187	י 203	ף 219	δ 235	√ 251
C	FF 12	28	, 44	< 60	L 76	\ 92	l 108	| 124	î 140	£ 156	¼ 172	¶ 188	כ 204	ץ 220	∮ 236	ⁿ 252
D	CR 13	29	- 45	= 61	M 77	] 93	m 109	} 125	ì 141	¥ 157	¡ 173	© 189	ל 205	§ 221	ϕ 237	² 253
E	14	30	. 46	> 62	N 78	^ 94	n 110	~ 126	Ä 142	ß 158	« 174	® 190	מ 206	∧ 222	∈ 238	³ 254
F	15	31	/ 47	? 63	O 79	_ 95	o 111	Δ 127	Å 143	ƒ 159	» 175	™ 191	נ 207	∞ 223	∩ 239	¯ 255

Table des motifs de remplissage

